Tuomas Paasonen

Methods for Improving the Maintainability of Application Software

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 13.4.2011

Thesis supervisor:

Prof. Seppo Ovaska

Thesis instructor:

D.Sc. (Tech.) Risto Sarvas

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tuomas Paasonen

Title: Methods for Improving the Maintainability of Application Software

Date: 13.4.2011 Language: English Number of pages:8+79

Department of Electrical Engineering

Professorship: Industrial Electronics Code: S-81

Supervisor: Prof. Seppo Ovaska

Instructor: D.Sc. (Tech.) Risto Sarvas

Society has become in many ways dependent on software. It runs several everyday tasks that we take for granted in modern life. The more society is in interaction with software the less the requirements of software systems can be isolated from the world around them. Changes of the operation environment have to be met with changes in the software.

Software maintenance is, therefore, not simply corrective changes. New features have to be constantly implemented to keep the software acceptable in the changing world. For a software with a long lifetime, maintenance can form more than a half of the costs. Improving software's maintainability not only reduces the costs but also prolongs the application's lifetime.

In this thesis, we search for methods that can be used to improve software's maintainability during its development phase. We do a case-study of three project audits in a mid-sized Finnish software company. The aim is to find common issues that affect software's maintainability.

We discover that maintainability is a sum of partly uncorrelated factors. It should thus be monitored through its components rather than one single measurement. Based on the project audits, we provide a list of general instructions that can be used as a guideline by development teams. As maintainability is to some extent subjective, the instructions cannot be used as such. Instead they have to be adapted to each company culture and project separately.

Keywords: Application Software, Software, Maintainability, Maintenance

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tuomas Paasonen

Työn nimi: Menetelmiä sovellusohjelmiston ylläpidettävyyden parantamiseksi

Päivämäärä: 13.4.2011 Kieli: Englanti Sivumäärä:8+79

Sähkötekniikan laitos

Professuuri: Teollisuuselektroniikka Koodi: S-81

Valvoja: Prof. Seppo Ovaska

Ohjaaja: TkT Risto Sarvas

Yhteiskunta on tullut monella tavalla riippuvaiseksi ohjelmistoista. Ne pyörittävät monia arkipäiväisiä rutiineja, joita me pidämme itsestäänselvyyksinä. Mitä enemmän ohjelmistot ovat yhteiskuntamme toimintoja, sitä vähemmän ohjelmistojen vaatimukset voidaan määritellä eristyksessä ympäröivästä maailmasta. Toimintaympäristön muutosten tulee näkyä myös muutoksina ohjelmistossa.

Ohjelmistojen ylläpito ei ole siis vain virheiden korjaamista. Uusia ominaisuuksia tarvitaan jatkuvasti, jotta ohjelmisto pysyisi hyväksyttävänä muuttuvassa ymäristössä. Pitkäikäiselle ohjelmistolle ylläpidon kustannukset voivat muodostaa jopa yli puolet kokonaiskustannuksista. Ylläpidettävyyden parantaminen ei ainoastaan pienennä kustannuksia vaan myös pidentää ohjelmiston elinikää.

Tässä diplomityössä etsimme keinoja, joilla ohjelmiston ylläpidettävyyttä voidaan parantaa jo kehitysvaiheessa. Teemme tapaustutkimuksen kolmesta projektiauditoinnista keskikokoisessa suomalaisessa ohjelmistoyrityksessä. Auditointien tavoitteena on löytää yleisiä ylläpidettävyyteen liittyviä tekijöitä.

Tulemme tulokseen, että ylläpidettävyys on useiden osittain toisistaan riippumattomien tekijöiden summa. Siksi sitä tulisi tarkastella osiensa kautta sen sijaan, että sitä mitattaisiin yhtenä kokonaisuutena. Projektiauditointien perusteella listaamme yleisiä ohjeita, joita kehitystiimit voivat käyttää suuntaviivoina. Koska ylläpidettävyys on jossain määrin subjektiivista, ei ohjeita voi käyttää sellaisenaan. Ne tulee sovittaa erikseen jokaiseen toimintaympäristöön ja projektiin.

Avainsanat: Sovellusohjelmisto, ohjelmisto, ylläpidettävyys, ylläpito

Preface

Programming has been a part of my life at least for the past fifteen years. What started with a few simple web pages in the late 1990's, has become a time-consuming hobby, as well as, a profession for me. In my work, I have been dealing with many projects in maintenance. Those have taught me the meaning of maintainability for any software company. Making this thesis has directed me to learn a whole lot more about the quality of software.

I want to give my thanks to Teemu, Olli and the whole Lifecycle Management team. They have given me their support and made it possible for me to work with this subject. I have been able to get opinions and feedback for my thesis, whenever needed from Elämänluola, also known as "the team premises".

I would also like to thank Eeva and my family for their support. A special thank you for my sister, Hanna-Reetta, who helped me with English and made a final check for the spelling.

Final thanks for the Ilkka Rämö and Risto Sarvas from Futurice as well as Seppo Ovaska who has supervised this thesis. They have given important and instructive feedback and instructions during the whole thesis process.

Espoo, 13.4.2012

Tuomas Paasonen

Contents

Abbrevations

API	Application Programming Interface
DoD	Definition of Done
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IP	Internet Protocol
UI	User Interface
UML	Unified Modeling Language

1 Introduction: Maintenance in Futurice

Software can be found almost everywhere in the modern society. It connects us to our friends and families even when they are on the other side of the world. It works mathematics when we are doing our grocery shopping. It wakes us up in the morning and makes our commute safer. If there is any technology around, software probably plays some part in it.

In many ways we have become even downright dependent on software and the services it controls. The more critical the service the more important it is that the software keeps running without problems. Errors need to be corrected as they occur, performance needs to be followed and improved if needed, new use cases might arise and thus require new features. In other words, software needs to be maintained.

Application software's lifetime might span from days to years. The longer it is available, the more need there is for maintenance effort. For a long living software, the maintenance effort might even be greater than the effort of the initial implementation. [1, 2, 3] Therefore, software maintenance holds a heavy financial load.

The financial potential always means a financial risk as well. If maintenance becomes a difficult task it will have a significant effect on the cost of products. As changes take longer and longer to make, money is spent in excessive amounts for even the most minor tasks. In addition, transferring maintenance to another vendor might become impossible. Thus customer would be unable to change an unsatisfactory vendor to another.

But it is not just the numbers that suffer from bad maintainability. In most cases, software errors are merely disturbing, but a failure in a critical system might cause accidents and true damage. Maintainability of a software depends on how easily errors can be recognized and repaired. Thus maintainability is an important factor in the safety of technology.

Even though maintainability is recognized as an important quality factor, almost any professional programmer can tell you a story or two about maintenance nightmares. When budget and schedules are tight it is often the quality work such as testing, documenting or refactoring that gets cut off first. More often than not, this leads to problems when the software is in production use.

Problems in maintainability are often forgotten and invisible until changes need to be done. Unlike many other software quality factors, maintainability is hard to measure or visualize. Regardless of numerous researches there are no commonly accepted criteria for it. [4, 5, 6] This is not unexpected as many intuitive factors of maintainability, such as an understanding of the language, depend on the maintainers themselves.

Because maintainability has such an important effect on product quality and budget, and because it is not intuitively defined, it is worth studying. As some factors of maintainability depend on the company culture or chosen technologies, it is reasonable to do this study in a rather small and specific environment.

This thesis was carried out at Futurice, a mid-sized Finnish software company. Futurice creates web and mobile applications for its customer companies. Many of the products done by them have a lifetime of several years. Thus it is important to understand and prepare for maintainability in this context.

In early 2011, a new maintenance team started at Futurice. The aim of the team was to improve maintenance's quality and profitability and to make it more interesting for the maintainers. A key element for each of the three goals was to improve maintainability of projects. The team faced two questions: how to evaluate and improve maintainability of products, and how to instruct developers to build maintainability into products from the beginning of projects.

These questions are the motivation for this thesis. To understand the factors of maintainability we first take a look at what others have said about it and how they have estimated maintainability. In the literature review, in Chapter 2, we define maintainability among other terms that are closely related to it. We also cover some ideas on how it could be estimated or measured. We also take a brief look at some methodologies that might help developers to tackle maintainability problems.

In Chapter 3, we define the actual research questions. To reflect the title of this thesis we ask how maintainability of application software can be improved. This question is divided into three sub-questions, which support one another to give a sufficient picture of maintainability in the context of Futurice. The sub-questions are: how can we estimate maintainability, which factors affect maintainability and what instructions to give development teams.

For the maintainability estimation three project audits were done. The audit process is described in Chapter 4. The background and the goals are explained further in that chapter to give the reader a better understanding of the environment. Describing the circumstances is also important for justifying the method itself.

In Chapter 5, we conclude the answers for the research questions. We find out that most is gained from discussions with the development team. Knowledge from automatic measurements and code base reviews is required to steer the discussion. This way the development team's silent knowledge can be made visible.

Maintainability factors on the code base differ somewhat from one language or framework to another. However, we conclude that there are some common rules as well as many practices that can be used to ensure improved maintainability. Consistency, which is gained through common agreements and automatic monitoring,

helps others to understand and change the code after the product is transferred to maintenance.

A few instructions are given as the summary of our results. These are high level ideas and rules of thumb that act rather as a guideline than an actual practice. The instructions are:

- *Consider maintenance from the beginning.* Maintainability cannot be built afterwards. Thus maintenance should be considered when making decisions concerning technology and quality standards.
- *Define and monitor "done".* The definition of "done" is a tool to commonly agree upon when a feature is accepted. It can help to keep the software quality high, if used consistently.
- *Automate processes.* Any software project includes some processes such as building, running test suite or migrating data from one version to another. Automating these processes eases maintenance as maintainers do not have to learn or remember each step of the various processes.
- *Share and store knowledge.* Understanding the software is a key element of maintainability. Continuous communications and sufficient documentation ensure that everybody has enough understanding to carry out their tasks in the project.
- *Test the code.* Testing cannot ensure that the code is correct. Instead, it can give a maintainer some confidence that new changes have not broken any old functionality.
- *Refactor along with changes.* The structure of code often requires updates when features are added or changed. Restructuring should be done right after changes when the developer already has the specific module in mind.

It must be determined separately in each project, how to implement these instructions in the particular project. As a subjective manner, good maintainability can probably be achieved with other practices, but these are the ones we found useful. Section 5.3 discusses them further.

2 Literature review: the why and how of maintainability

To understand maintainability we must first understand what software and its maintenance is. Software's lifecycle and evolution are essential for this thesis. Those must also be discussed before researching maintainability itself.

In this chapter, we define the most important terms and phenomena concerning maintainability. Then we will introduce how others have researched maintainability. We will describe factors, metrics and methodologies that are connected with maintenance and maintainability.

2.1 Application software defined

Grubb and Takang define software as "the programs, documentation and operating procedures by which computers can be made useful to man". [1] It follows largely the definition by IEEE [7] but is clearer and shorter in its form. In particular, it emphasises operating procedures as part of a software. We will be using this very wide definition of software throughout this thesis.

We research software maintainability but restrict the scope to application software. The latter is defined by the IEEE Standard Glossary of Software Engineering Terminology as "Software designed to fulfill specific needs of a user; for example, software for navigation, payroll, or process control." This is in contrast to system or support software which either facilitate the computer resources or aid in developing or compiling other software. [7]

Another definition that is closely related to application software is given in SPE-classification. It divides all software in three different categories. These categories are summarized in the following paragraphs and in Table 1.

S-programs solve problems that can be defined in a specific manner. For example they might compute a solution to a mathematical problem or generate a specific tone with some predefined speaker. Even though S-programs might simulate some real world phenomena they operate in their own abstraction universe that can be fully defined. If the specification of an S-type software changes, the changed solution has to be a new software.

P-programs differ from S-programs in that they approximate the real world rather than define their own universe. This means that where S-type software's correctness depends only on how they fulfill their specification, P-type software's solution must be compared to the real world and evaluated in the context in which they are intended to approximate. An example of P-program would be a weather prediction system. It could, in theory, be defined precisely, but the Earth's weather system is such a complex phenomena that many aspects of it must be assumed, and

thus solutions are approximations rather than exact values.

E-type software is described as "programs that mechanize a human or societal activity." It is the one most involved in the real world of all the program types. E-programs operate as a part of the outside world, rather than trying to just simulate it. Like the world around them, the E-type applications themselves are prone to change. Their validity depends on the human perspective of users. An E-program might be useful even if it does not fully satisfy its specifications. On the other hand, it might be useless even though the implementation is done properly. [8]

Table 1: Comparison of SPE-classification programs.

Type	**Compares to real world**	**Validity**
S	Predefined universe	Fully implements specification
P	Approximation of real world	Comparison to real world is satisfiable
E	Embedded in real world	If useful from users' perspectives

As we can see the E-type is closely related to application software although it is not fully equal with it. In this thesis, we will nevertheless restrict the term application software to cover only E-type software. This is due to large theoretical bases of the E-type software's evolution that is covered more closely in Chapter 2.2.1.

2.2 Maintenance in the software lifecycle

The software lifecycle is the time span that begins the moment the product is conceived and spans until the moment it is no longer available to use. The lifecycle of a software consists of different phases that may change from one application to another. In the usual case, the phases are specification, design, implementation, testing, installation, and maintenance. [7] In Figure 1, a waterfall model of software evolution is presented. In the waterfall all the phases are done one after another. All these phases may also be ongoing simultaneously. Figure 2 shows an example of this kind of software lifecycle.

In this section, we describe software evolution to understand the requirements for maintenance work. In the last part we define what we mean by "maintenance" in this thesis and how it positions itself to the software lifecycle.

2.2.1 Software evolution

During its lifetime, an E-type software and its correctness changes with the changes in the environment it operates in. These changes are called "software evolution."

Software evolution has been researched since the late 1960's, although the name was proposed by Les Belady in 1974. [9] To justify the need for maintenance and

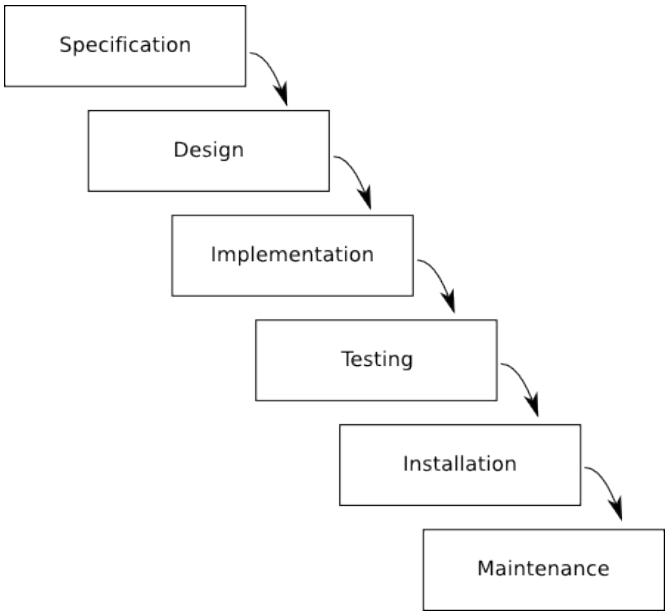

Figure 1: The waterfall model of software evolution.

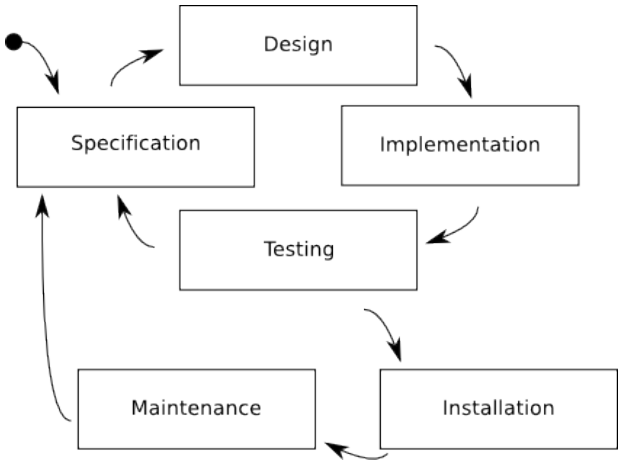

Figure 2: An iterative model for software evolution.

thus maintainability, it is important to understand how software evolution works. The theory of software evolution shows that even if the original implementation is correct and fully covers the requirements, the software has to be updated regularly for it to stay satisfactory.

In 1970, Cambell proposed a model of the number of errors that appear after the original release of a software. The model is presented in Figure 3. Cambell argues that because old errors often reoccur in future releases and users become more familiar with the product, the count of errors found in a software product increases when it ages. [10]

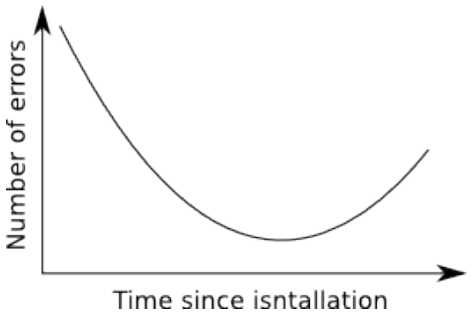

Figure 3: The number of errors as a function of time since the release.

2.2.2 Lehman's laws

The theory of software evolution is written out in Lehman's eight laws. They describe the phenomena that cause the software to lose its importance over time, if no improvements are done. The following list [11] contains the eight laws and their publishing years.

1. The law of **Continuing Change** (1974): E-type systems must be continually adapted or they become less satisfactory.

2. The law of **Increasing Complexity** (1974): As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it.

3. The law of **Self Regulation** (1974): E-type system evolution process is self regulating with distribution of product and process measures close to normal.

4. The law of **Conservation of Organisational Stability** (1980): The average global activity rate in an evolving E-type system is invariant over product lifetime.

5. The law of **Conservation of Familiarity** (1980): As an E-type system evolves all people associated with it e.g. developers, sales personnel, users, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

6. The law of **Continuing Growth** (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

7. The law of **Declining Quality** (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

8. The law of **Feedback System** (1996): E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

The laws 1, 3, 6 and 7 incline that the software becomes less satisfactory and its quality will decline over time. This is not because the software itself is changing, but because the expectations upon which it has been designed and implemented are changing. A few examples of these changing expectations might be:

- The environment in which the application is used changes. [1]
- The competitive products raise the expectations of users. [1]
- Customers require new features as new use cases arise.

The second law implies that if there is no work done to specifically lower the complexity of the software it will grow as new features are added. Increasing complexity leads to increasing difficulty of making changes. This means that the effort estimates for tasks will increase over time. Either they include work for reducing complexity, or the development time of new features will increase. [1]

The laws 4 and 5 are closely related. They state that over time, productivity will remain constant and that the speed of growth will be constant or decline. This is due the increasing complexity or extra effort for simplifications. The important point here is that profitability cannot be increased by deciding to use less time on quality. [1] The fifth law states that it is the complexity of the software that prevents the acceleration of growth. This implies that simplification is the key for steady development.

The eighth law is conclusive of the other laws. It states that the development of an E-type software is dominated by the feedback from the surrounding world. [1] This is in the very core of E-type software and thus also application software. See Section 2.1.

Another theory closely related to Lehman's laws is the principle of software uncertainty. It states: "Even if the outcome of past executions of an E- type program have been satisfactory, the outcome of further executions is inherently uncertain; that is, a program may display unsatisfactory behaviour or invalid results." [9] Due to the feedback systems in Lehman's laws, the application's output might become less satisfactory in time even though the software itself does not change. This concludes what all eight laws together imply.

2.2.3 Maintenance

In the traditional waterfall model maintenance is the last phase of a software's lifecycle. It begins after the product is implemented and tested and it continues as long as the software is available for use. [1] In the waterfall, this would mean that maintenance is actions to correct errors in software that got past the testing phase.

Taking into count Lehman's laws, we can see that more than just corrections need to be done after the initial release. As demands change, so must the product.

This is why maintenance is not just corrective acts but also includes the effort to adapt the software to new requirements by adding or improving features.

The third aspect of maintenance is making changes that aim to increase maintainability, performance or other metrics. [12] They differ from corrective and incremental changes in that they do not change the observable behaviour. In here, we will call all of these changes "improvements". We have noticed in practice that these improvements are, in many cases, important for maintenance work. Customers often see them as a burden, as they increase costs but do not implement new functionality. The different parts of maintenance work are presented in Figure 4.

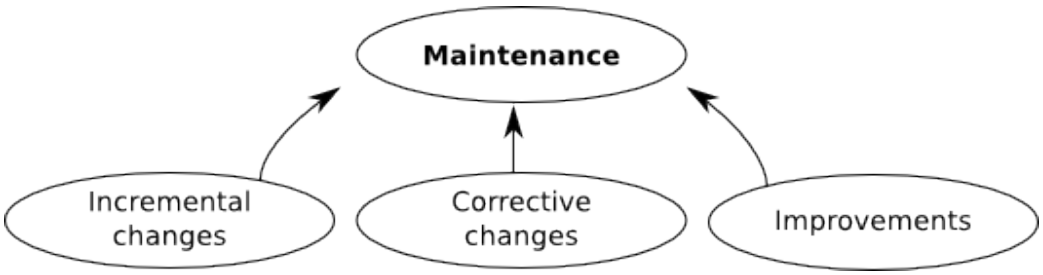

Figure 4: Maintenance consists of changes and improvements.

Even though new features are added in the maintenance phase it cannot be described simply as continuation for new development. There is a difference in constraints between the development and maintenance phases. The existing system sets its limits on the type of changes and how they can be carried out during maintenance. [1]

It is indicated by many researches that a significant portion of the total effort in a software project is spent on maintenance. The percentage differs depending on the product but on a large-scale project, maintenance effort can be half or more of the total effort. [1, 2, 3]

2.3 Maintainability

2.3.1 Definitions and metrics

"Maintainability" is a notoriously difficult word to define. The problematic nature of the term can be seen from the IEEE definition: "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment." [7] The definition in all its simplicity is sufficient and intuitive, but it lacks the accuracy that is often required.

As was mentioned in Section 2.2.3, maintenance can cost more than half of the project effort. There is a lot to gain financially from improving the maintainability of a software and thus making the maintenance process easier. This is why many

studies suggest ways to measure or estimate maintainability as a numeric value. This kind of maintainability index could be computed with analytical tools that automatically calculate maintainability factors like test coverage, lines of code in file or comment lines per lines of code. There is no one commonly accepted quantitative measure for maintainability available, but we will cover some ideas here.

To measure maintainability it has to be redefined in a manner that has some mathematical context. Pfleeger suggests that maintainability is a probability that maintenance task can be done in a certain amount of time. [5] With this approach maintainability could be measured as a statistical feature. We see two problems in this definition. Firstly, it does not tell us what kind of problems the application has, it merely estimates the time it takes to solve them in an average case. Secondly, the definition is hard to apply on time estimates because maintenance tasks are often hard to specify in beforehand.

Another way to see maintainability is to split it up in smaller entities. After this maintainability can be measured indirectly through these factors. The methods can be divided into structural measurements and expert assessments. [4] For structural analysis, there are several different metrics and tools available. Metrics used for maintainability estimation include both very low-level statistical calculation and more advanced theories of software quality or complexity. Some examples of the metrics used are: lines of code per lines of comment, [6] number of methods in class and coupling between objects. [4]

Expert assessments use either very strictly defined instructions for evaluation or follow some guidelines of what may decrease maintainability. Fowler and Beck have introduced a list of code qualities that can be used as an aid for unguided assessment. [4] These qualities are discussed further in Section 2.7.

As the maintainability measurements use indirect metrics, their results must be summed up with proper weights. Different models for maintainability metrics usually answer two questions: which metrics are taken into count and how the weights are set. In recent studies, a fuzzy logic model is usually used to combine measurements or expert's estimations. [5, 6, 13]

Land takes another approach for measuring maintainability. He presents a model where the maintainability at a particular moment is not important. Instead, he tries to present in which direction the maintainability is going. [5]

Neither structural measurements nor expert assessments have been proved sufficient empirically. [4, 14] Mäntylä has also shown that opinions of expert often collide with each other and with structural measurements. [14]

2.3.2 Maintainability factors

Various kinds of definitions and metrics for maintainability has been described in Section 2.3.1. The aim of this thesis is to describe ways to improve maintainability not necessarily measure it in exact form. This is why a very wide definition can be used. For this thesis, maintainability is defined as follows:

- **Maintainability:** The ease of making a change to software.

As was mentioned Section 2.1, software is more than just the source code. We will focus not only on the code, but also on documentation and testing as well as procedures that are used for installing, using, or developing the software. These are all factors that affect the maintenance work.

Yang and Ward say that "Possibly the most important factor that affects maintainability is planning for maintainability." [15] This means that maintainability has to be built inside a project during the development phase or it will be very difficult if not impossible to add afterwards. Therefore estimation and improvements need to be done continuously starting from the beginning of the project. The process of improvements is called refactoring and is discussed in Section 2.5.

This thesis will cover the most important factors that affect software maintainability and propose ideas and instructions for developers on how to improve the maintainability during the development phase. As there is no commonly agreed measurement for maintainability, we will not propose any one measurement to be used to recognize maintainability problems. Instead we propose that multiple maintainability factors should be measured separately to identify problem points.

2.4 Technical debt

The term "technical debt" is increasingly used in the software industry. It is a metaphor that describes how quality work is left undone during the development phase to meet the customer's time or budget expectations. [16] Technical debt can consist, for example, of untested code, code smells (see Section 2.6) or conventions that were not followed.

In this thesis, technical debt is viewed from the perspective of maintainability. Therefore we define it to consist all the technical aspects that affect the ease of change.

- **Technical debt:** The technical features of software that decrease maintainability and other quality aspects.

Guo and Seaman argue that this metaphor is closely similar to financial debt. Even though quick profits are gained when technical debt is increased, in the long run the debt will grow interest. In their recent study (2011), they followed a single

delayed maintenance task and evaluated the cost of the delay. They found out that delaying this particular task almost tripled the cost of a later change in the software. [16]

Unlike financial debt, technical debt is not necessarily paid back. It can stay in the code base for the rest of the product's lifecycle. Due to technical debt, the complexity of the product will grow and productivity decrease. This phenomenon can be seen from Lehman's laws that are described in Section 2.2.2.

In their study, Guo and Seaman talk about a single task that was delayed by manger's decision. [16] Technical debt can also grow silently without any specific decision being made. It is increased any time a developer leaves some improvements to be done later.

2.5 Refactoring

Refactoring is a term that does not have one commonly agreed-on definition. Fowler presents two possible definitions that are close to each other. The first definition is "to restructure software by applying a series of refactorings without changing its observable behaviour". The second one is a bit more precise: "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour". [12]

As we can see, these definitions have a lot in common points. Particularly the last part in both is that the change does not affect the observable behaviour of application. So clearly, refactoring is not a corrective or incremental change to application. It does not increase the business value of an application, but aims solely to improve maintainability. This is demonstrated in Figure 5.

Another thing they both share is changing the structure. This is in contrast with, for instance, performance improvements. They usually do not have an effect on behaviour but the goal is different than with refactoring.

For this thesis, the keywords are in the latter definition. They say that refactoring aims to make software easier to understand and cheaper to modify. Hence refactoring is the process of improving maintainability by reducing technical debt. We use this aspect and the previous definition of maintainability (see Section 2.3.2) to define refactoring.

- **Refactoring:** To restructure software for reducing technical debt and improving maintainability without changing its observable behaviour.

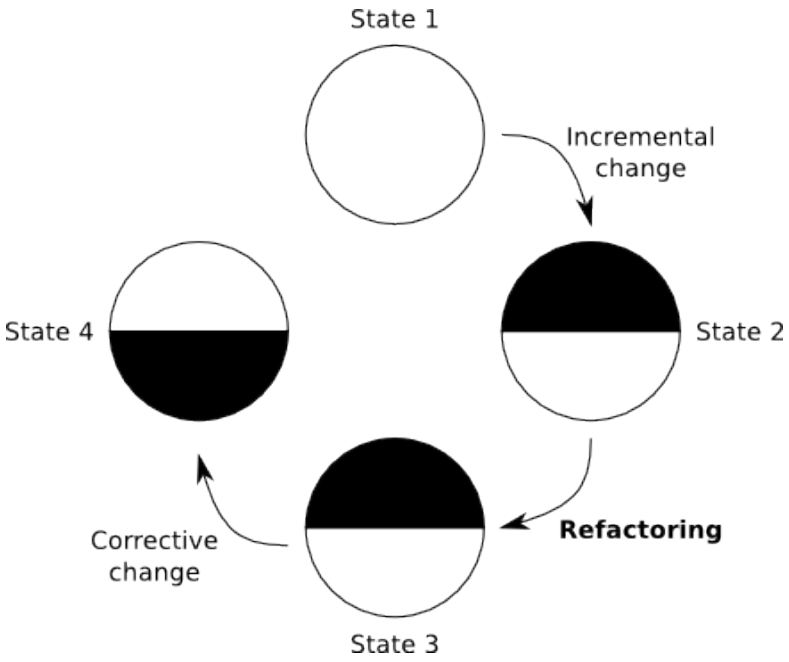

Figure 5: Unlike corrective or incremental changes, refactoring does not change the observable behaviour.

2.5.1 When to refactor?

McConell argues that 80 per cent of the benefits can be achieved with just 20 per cent of refactoring. This is why a developer should be aware of when and where to do refactoring. According to this 80/20 rule, even small changes might be enough to improve maintainability significantly. [17] Although McConell does not offer any proof for this claim, we argue that this is a good rule of thumb. As different maintainability factors cause problems of different magnitudes, improvements offered by refactoring also depend on the type of refactoring done.

Fowler and Beck define signs of bad code which indicate a need for refactoring. They call the signs smells in code [12], and they are presented in Section 2.7. But refactoring is not always the best approach even if the code is smelly. McConell introduces some ideas on when to refactor and when not to refactor.

If the code is not working repairing it is not refactoring. Making the structures more simple might be a step in fixing a bug but it is not refactoring. The definition clearly states that it does not change the behaviour of application. Thus a developer should not mask corrective changes behind refactoring.

McConell argues that sometimes it is more appropriate to rewrite the code than just refactor it. Major refactoring effort usually means that the whole section should be redesigned in another manner. Without a proper rewrite, the result might be worse than the original implementation.

In maintenance phase, code sections that are not touched are usually not worth refactoring. But when the section is to be modified it should be reviewed and refactoring should be done if the code is not clean (see Section 2.6). Refactoring changed sections right after the change is also part of Feather's algorithm for change. [18]

Refactoring should be done as early as in the development phase. As the application is built it changes and some designs might become deprecated. Refactoring should be considered when adding a routine, adding a class or fixing an error. Also if there are modules that are known to be error-prone or very complex they should be refactored. [17]

Choosing when to refactor is a question on profitability. The aim of refactoring should be to improve maintainability, without spending the gained advantage for improving details that have little real value. Therefore refactoring should be done right after incremental or corrective changes, when developer is already familiar with the piece of code to be refactored.

2.5.2 How to refactor?

Although there are guidelines on how refactoring can be done, it is clear that refactoring is just an act of programming. In this section, we will look at ideas of the Fowler and Feathers on the subject. Feathers describes an algorithm for making a change, and Fowler proposes practical instructions on actual refactoring tasks.

The definition of refactoring emphasises that there should be no change in observable behaviour. But when code is restructured, there is always a risk of unforeseen consequences. Both Fowler and Feathers argue that testing is thus a crucial part of refactoring. It does not ensure that there are no errors, but at least it provides the chance to notice when something is broken. The importance is that developers have more confidence for refactoring.

Feathers describes how to change the code that is in its maintenance phase. He includes refactoring as a part of the change process, thus promoting continuous improvements to quality.

Feathers' algorithm begins with identifying the points that need to be changed. When the components are found, the influenced code needs to be isolated from dependencies so that change would not affect the component. The isolated code is then covered with unit tests.

Only after the code is isolated and tested can it be changed. When the implementation is done, the developer has an understanding of the component and the influenced part of the code is covered with tests. Therefore, this is an opportunity to refactor and make any future changes easier. [19]

Also, Fowler argues for the importance of testing in refactoring. Fowler says that refactoring, in fact, requires testing. More discussion on testing is provided in Section 2.9.

Fowler also presents many methods for refactoring. It is not in the scope of this thesis to describe these methods in detail, nor even list them all. A few examples include the *Pull Up* method, which means that a method common for multiple subclasses should be implemented in a common parent; the *Remove Middle Man* method for making two classes to discuss without another class in the middle; and the *Replace Data Value with Object* method to avoid duplicate code or feature envy. [12]

2.6 Clean code

To make a change in the software, a maintainer needs to be able to understand the code that was written by the original developer. It is commonly understood that the more complex, scattered and messy the code is, the harder it is to understand and maintain it. [20, 19, 21, 10] In sections 2.6 and 2.7, we will discuss several subjects that concern the qualities of source code that affect maintainability.

This section borrows its name from a book by Robert C. Martin. He presents ideas on how to make source code clean and readable. We will not discuss all the subjects in his book here, but we give a picture of the kind of properties does clean code has.

Many of the subjects in Section 2.6.2 are preferences of the various authors we will be citing. As Martin says, code styles are more a question of taste than absolute truths. But as is explained in Section 2.6.1, although some particular practices might be controversial, having practices in general is not.

2.6.1 Code conventions

Convention is described by the Oxford English dictionary as "a rule or practice based upon general consent". [22] Coding conventions, too, are practices or rules commonly agreed on by the developers in a project. They do not affect functionality but they might make the code more readable and easier to maintain.

Many decisions in programming are rather arbitrary. Choosing a bad name for a variable does not prevent the code from running. Neither does naming all the functions in a different style. But both decrease the readability of code.

Martin argues that programmers should use conventions as widely agreed-on in the industry as possible. [21] McConell regards conventions as tools for managing the complexity of the code. He says that they prevent programmers from making

arbitrary decisions, such as naming or indentation, inconsistently. [17]

McConell lists various advantages in using conventions. He argues that they can help developers avoid hazardous practices, bring predictability to low-level routines or compensate for weaknesses of language. He says though that too many conventions can be hard to remember and thus become a burden. [17]

2.6.2 Good practices

Although choosing code conventions is an arbitrary task, there are some common guidelines to be followed. Numerous books have been written on coding style [17, 21, 12] and it is not in the scope of this thesis to cover them fully. Instead, a brief look at the subject is given here by listing some of the most important practices.

The practices mentioned here are just a few examples of coding standards that could be taken into use. When a project is started development team should agree on common practices to follow. When consistently used, good practices will make the code more readable. [17, 21]

Naming is considered to be one of the key features of clean code. McConnell argues that, unlike for example the names of pets the names, in code are basically the same thing as instances themselves. A good name should describe what the instance is, or the instance itself becomes useless. [17]

The importance of good naming is demonstrated in Listing 1. As can be seen here, just by writing meaningful names for variables and functions, a developer can give code more structure. Even short functions that are rather easy to understand with meaningless names, benefit from proper names, as with them a developer can read the code almost like any other text.

Listing 1: The same function with two different naming conventions

```
1  // An example with short variable and function names
2  int mltp(int a, int b) {
3      int c = a * b;
4      return c;
5  }
6
7  // The same code with meaningful names
8  int multiply(int original, int multiplier) {
9      int result = original * multiplier;
10     return result;
11 }
```

Short functions. Each function should only do one thing as simply as possible. Defining "one thing" though is not easy to do. A more accurate instruction is given by Martin. He says that functions should not be longer than a few lines. Long functions easily become more complex and harder to understand. [21] Function size is also described as one of the code smells, and is discussed further in the Section 2.7.1.

Short functions do not have the space to do many things. If it seems that what was assumed one thing does not fit within few lines, it should be split into smaller functions. An example of this is given in Listings 2 and 3.

Both implementations transform an array of strings into an HTML list. There are four required steps: decoding HTML special characters, adding HTML tags for list items, combining items into one string, and adding tags for the list wrapper. All of these steps are done in one function in the Listing 2.

Listing 2: An example of a function that does more than one thing.

```
String arrayToHtmlList(String[] items) {
        String innerHtml = "";
        for (String item: items) {
                item = item.replaceAll("<", "&lt;");
                item = item.teplaceAll(">", "&gt;");
                item = item.replaceAll("&", "&amp;");
                item = "<li>" + item + "</li>";
                innerHtml += item;
        }
        return "<ul>" + innerHtml + "</ul>";
}
```

The same functionality is found in Listing 3, but now in three different functions. The advantages gained here are separating the combining of elements and decoding them. For example, if more rules for decoding would be required it would be easier to find from a separate function. Also this functionality would be easier to debug and test.

The function *arrayToHtmlList* still implements two of the four steps. This is to show that each line should not have their own wrapper function just because it makes functions shorter. "One thing" could be defined here as "combining items into HTML list".

Listing 3: The logic of Listing 2 split into three smaller functions

```
String arrayToHtmlList(String[] items) {
        String innerHtml = "";
        for (String item: items) {
                innerHtml += decorateAsListElement(item);
        }
        return "<ul>" + innerHtml + "</ul>";
```

```
7  }
8
9  String decorateAsListElement(String item) {
10         String innerHtml = decodeHtmlTags(item);
11         return "<li>" + innerHtml + "</li>";
12 }
13
14 String decodeHtmlTags(String item) {
15         String result = item;
16         result = result.replaceAll("<", "&lt;");
17         result = result.teplaceAll(">", "&gt;");
18         result = result.replaceAll("&", "&amp;");
19         return result;
20 }
```

Only meaningful comments. Martin argues that most comments could be replaced with proper sized and properly named functions and variables. The problem with having a lot of comments is that they easily get out-of-date and only serve misinformation after that. Not all comments are bad, however. If used carefully, they have a potential for improving the readability of the code. [21]

Listing 4 shows a few examples of comments that are unnecessary as the variable or function names already describe the same information. Removing these comments would not only remove duplicate information, but also make the source shorter and easier to read through.

Listing 4: A few examples of unnecessary comments.

```
1  // Number of items
2  int numberOfItems;
3
4  // List of names
5  List<String> names;
6
7  // Remove commas from string
8  String removeCommas(String original) {
9          // [No implementation presented here]
10 }
```

Defencing programming has gotten its name from defencive driving. It is a style where the developer makes sure her unit of code will work correctly or throw an error even if the input from another unit is not satisfactory. A developer thus makes sure that at least that particular piece of code will not cause others to fail. [17]

Efficient logging is helpful when trying to find and correct an error. A good logging includes description of the error and the exact conditions on when the error occured. [17] Conditions include e.g. time, date, user identification, client application or such.

2.6.3 Complexity

Complexity is brought up many times in this chapter. It is intuitive to see why. If a piece of code is less complex, it is easier to understand and thus easier to maintain. McConell, with two examples from Hewlett-Packard and Construx Software, shows that, in these cases, the applications with fewer complexity also had less errors and problems.

Although complexity differs from other subsections of clean code (2.6) it, or rather avoiding it, is often the reasoning behind good practices. McConell argues that programming is a very complex task in itself. A developer has to keep in mind many routines and variables at a time in order to make the algorithm work. There is little chance for improving human capability but there are ways to reduce the complexity and thus make programming easier. [17]

There are many factors that affect code's complexity. McCabe introduced a complexity measure in 1976. It measures the complexity of control flow in a program. [23] Although McCabe's model does not cover all the aspects, it is commonly acknowledged that control flow is one of the biggest factors in code complexity.

One issue affecting complexity is the coupling of modules. It has been proposed that the complexity of a component-based system could be computed with a coupling measurement. [24] This shows that complexity can increase on control structures but also on how the components interact.

Other factors include the number of variables, the variables' lifetime, and the number of inputs and outputs. In other words the more variables or structures you have to keep in mind while programming a single unit the more complex your code is. To reduce complexity, an algorithm or task should be divided into smaller pieces and into multiple functions or methods. [17]

Although complexity is not a direct opposite of maintainability, it definitely makes code harder to understand. Complexity is also easy to trace, as it can be computed with many measures. Because of this it is an effective way to detect and reduce bad maintainability, especially when both control structures and coupling are taken into account.

2.7 Bad smells

Bad smell is a term proposed by Fowler and Beck in the book "Refactoring". [12] Unlike clean code bad smells are based on human intuition. They are more general than just coding style. Smelly code is not always the same as bad code, but it is an indication of possible problems. The original idea of smells is to give developers pointers on when to refactor their code. Refactoring is explained more in Section 2.5.

The book "Refactoring" is written with Java in mind. The ideas are not exclusively for Java, but most of them are exclusively for object-oriented languages. We do not separate the smells according to their compatibility with different languages but notice that not all of them are relevant for procedural programs. We use the same terms, methods and classes, as Fowler. In many cases, they could also be called functions, procedures or other entities.

Mäntylä, Vanhanen and Lassenius introduced a taxonomy for the original smells. The purpose of the taxonomy is to make the smells more understandable and show the relationships between them. [25] We will use this taxonomy in this section to discuss the smells.

2.7.1 Bloaters

- Long method
- Large class
- Primitive obsession
- Long parameter list
- Data clumps

Bloaters endanger the readability of code. Some of the bloaters are by definition long sections and other are usually seen as a cause for such. [25] Fowler and Beck argue that long methods are difficult to understand and thus changes made to them are harder than the changes made to shorter methods. They also say that large classes have often too many instance variables which easily lead to duplicate code. Large methods and classes should be divided into smaller entities. [12]

Long parameter lists easily become inconsistent. When the code is changed, they tend to gather more and more parameters. This makes them more difficult to understand and use. [12]

Primitive obsession, the usage of multiple primitive values instead of structure class and data clumps, a group of primitive variables that can be found from many classes, [12] are not bloaters themselves. However, primitive obsession causes large classes and data clumps case long parameter lists. This is why they are listed along with bloaters. [25]

2.7.2 Object-orientation abusers

- Switch statements
- Temporary fields
- Refused bequest
- Alternative classes with different interfaces
- Parallel inheritance hierarchies

The name of this group is rather self-explanatory. These smells indicate that the code is not done by the guidelines of object-oriented programming. Either their inheritance is unnecessarily complex or they do not fully take advantage of the possibilities of object-orientation. [25]

Switch statements in object-oriented applications are usually used for recognising the class of an object. Fowler and Beck say they should be replaced by inheritance in such cases. Temporary fields are instance variables that are not always required by the class. These might be confusing for a developer who is trying to understand the functionality of a class.

Refused bequest means that a child class inherits unnecessary methods from its parent. Usually this smell is not strong enough to be worth cleaning. Another smell concerning inheritance is parallel inheritance hierarchies. This means that adding a new subclass for one class requires a new subclass for another class.

Alternative classes with different interfaces cause confusion. If two classes perform similar jobs, they should be named accordingly. [12] This idea follows the conceptual integrity described in Section 2.8.1. [10]

2.7.3 Change preventers

- Divergent change
- Shotgun surgery

Although all the smells are signs of decreased maintainability, the smells in this group increase the complexity of change significantly. They introduce two opposite problems that entangle concepts and structures.

In principle, a change in an object-oriented program should only affect one class. Let us consider an example where two changes are made in a software. The first one is adding a new field to database, and the other one is adding a new algorithm to the application.

We introduce two classes for our example: DatabaseAbstraction and ToolWithMultipleAlgorithms. The inner logic of the application should be separated so that the two changes are done in different classes. Adding a field should only affect DatabaseAbstraction and adding an algorithm should be done inside ToolWithMultipleAlgortihms only.

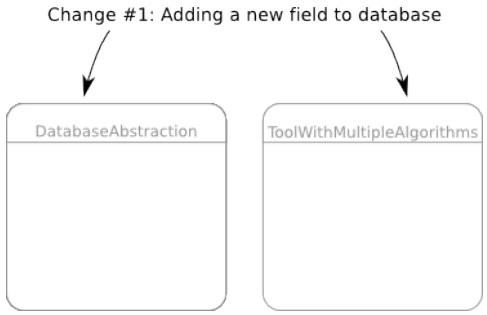

Figure 6: Illustration of shotgun surgery.

If the separation of classes is not done properly two situations might arise. If adding a field to the database requires changes in both classes updating ToolWithMultipleAlgorithms is easily forgotten. This is called shotgun surgery and it is presented in Figure 6.

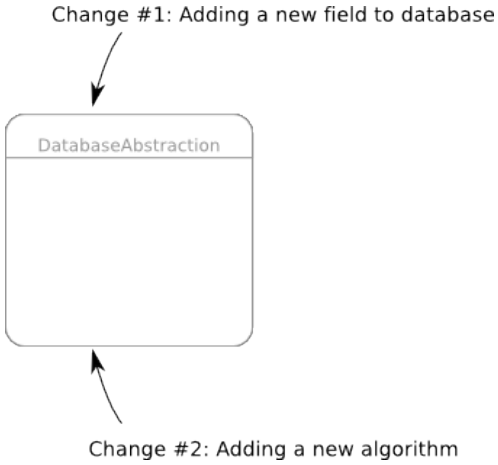

Figure 7: An illustration of divergent change.

Divergent change means a situation when both changes need to be implemented in either classes. Figure 7 is an illustration of this. Adding a field to database as well as adding an algorithm cause changes in DatabaseAbstraction. This makes change more complex and thus reduces maintainability. [12]

It is easy to see here that entangled modules cause a decrease in maintainability, although we argue that these smells are not always caused by the development team

but by the framework they are using. On the other hand, a well designed framework can also reduce the need for Change preventers, as it can make the changes to the different components on its own.

2.7.4 Dispensable

- Lazy class
- Data class
- Duplicate code
- Speculative generality

This group consists of smells that indicate code that should be removed. [25] Data classes only have getter and setter methods for their primitive variables. According to Fowler, this usually means that the data is being manipulated outside the class. Lazy class is, in a way, similar to Data class. It does not have enough functionality to be worth having its own class. [12] Duplicate code is intuitively understood as a piece of code stored in two different places.

Speculative generality means that a class has too many methods. It differs from the Large class (see Section 2.7.1) in that it is not about the lines of code, but about the features required to fulfill the specifications. Consider, for example, a class called Person. It stores, among other data, people's birthdays. The application only shows birthdays to users as dates, but the Person class also includes the getAge-method that computes the age in years. As this method is not required but implemented "just in case", it should be removed. [12]

Mäntylä, Vanhanen, and Lassenius propose a new smell, dead code, that was not used by Fowler and Beck. Dead code is code that is never executed. In contrast to speculative generality, dead code has been used in the earlier versions of the software but has later become deprecated. [25]

We argue that lazy and data classes are sometimes hard to avoid. Duplicate code and speculative generality instead are common and unnecessary. Removing features that are not used makes the code base simpler and thus easier to handle. Dead code is an important addition to code smells too, as it is hard to recognize and remove in the maintenance phase.

2.7.5 Encapsulators

- Message chains
- Middle man

Encapsulators are smells that consider object interaction. Message chains are a situation where one class asks an object for another object which asks for another object and so on. [12] An example of message chain is presented in Listing 5.

Mäntylä, Vanhanen and Lassenius argue that reducing this smell often causes for middle man to increase. [25] This means that a class passes data from another class through its own method without changing it in any way. [12] An example of middle man is found in Listing 5.

Listing 5: Examples of message chains and middle men

```
1  // Message chain
2  String input = InputWrapper
3                  .getStringInputStream()
4                  .getStringInputStreamReader()
5                  .getLine();
6
7  // Middle man
8  String getLine() {
9          return this.reader.getLine();
10 }
```

Where message chains do not take advantage of the possibilities of the object-oriented approach, middle men use them too much. Fowler and Beck emphasize that encapsulation in itself is a good thing but when two classes could talk to each other, no middle man should be added to increase the complexity. [12]

As reducing one encapsulator increases the other, fixing either one is a balancing act. In many cases, encapsulation is a good way to separate different entities from each other, thus we argue that especially middle men have their advantages when used carefully.

2.7.6 Couplers

- Feature envy
- Inappropriate intimacy

One important principle of object-oriented programming is that classes should be independent. They should concentrate on one thing and do it independently from other classes. Feature envy and inappropriate intimacy both increase coupling of classes.

Feature envy means that one class uses too many methods from another class. [12] Defining “too many” is rather arbitrary, as no class can work in total isolation. It is up to the developers to decide if a piece of logic which requires a lot of support

from another class should be placed in the envied class instead.

Inappropriate intimacy occurs when two classes use each others' methods or variables excessively. [12] This makes the classes tightly coupled on each other thus breaking their independence. Just like with feature envy, the limit between normal interaction and inappropriate intimacy is a matter of preference.

2.7.7 Others

- Incomplete library class
- Comments

The last two smells are grouped into Others for the lack of a better group for them. [25] The first one has to do with third-party libraries, and the second is more about coding style than the code itself.

Incomplete library class means that a developer needs some unimplemented functionality from a library class, but is unable to change the class. This is why library classes often require a wrapper class. Features that are missing from the third-party code can then be implemented into the wrapper, making the library look more complete for the rest of the software.

Comments is more controversial than other smells. They are not bad themselves but Fowler and Beck argue that they are often used as a deodorant, a kind of excuse for bad code. [12] This same idea is also discussed by others, and was presented earlier in Section 2.6.2.

2.7.8 Discussion of bad smells

In their later work Mäntylä, Vanhanen and Lassenius have discussed bad smells as a tool for evaluating code quality and maintainability. [14] In their empirical study of bad smells, they conclude that results depend highly on who is using the smells. People tend to see code differently, and thus the result of expert reviews are subjective.

Another finding in the empirical study was that expert estimations of bad smell do not correlate with evaluations made by code metrics. The comparison was made on large class, long parameter list and duplicate code as they were easy to measure with metric tools. [14] This enforces the subjectivity of expert reviews. On the other hand, code metrics themselves offer a rather narrow view on the code. Hence, they cannot be fully compared to expert reviews.

The subjectivity of expert reviews can be seen as a problem. If maintainability is not understood similarly by developers it is hard to justify refactoring. However,

it is probable that developers see the same smells but their estimation of the urgency differs. It also has to be mentioned, that while expert reviews are subjective so is maintainability itself. "The ease of change" depends to some extent on the experience of maintainers.

2.8 Consistent systems

Although bad smells in Section 2.7 describe some aspects of code that go further than individual units, most of the discussion this far has been about coding style or single line decisions. Many factors of maintainability are in how the whole system is designed. Therefore, it is important to discuss how to build consistent systems that are easy to understand.

2.8.1 Conceptual integrity

Conceptual integrity is closely related to code conventions presented in Section 2.6.1. It takes the ideas of conventions to system level. The term was presented by Brooks in his book "The Mythical Man-Month".

The idea of conceptual integrity means that the design decisions are consistent throughout the system. Brooks argues that it is the most important factor of system design. The whole system should reflect one conceptual idea.

Brooks introduces an idea where design decisions would be made by a single developer. He proposes that development teams organize themselves into what he calls "surgical teams". In such a team, the actual act of coding is only performed by one programmer. Everyone else supports him by documenting, testing or solving obstacles. [10] The idea reflects the programming of the 1970's, but provides the basic idea of sustainable system design.

If every piece of the code is built differently it is very hard to understand the relations between different parts. This makes changes harder to do and thus decreases maintainability. But it is hard to imagine that in a modern environment this idea could work as is. Instead, conceptual integrity should be built through architectural design where each developer has their say. Also, common conventions are important for making unified software.

2.8.2 Continuous integration

Integration is putting all the pieces together. It might mean making a whole program from individual modules or classes, or it might mean sewing together multiple programs to build a software system. McConell argues that integration is one of the most time-consuming phases of software development. Badly done integration can lead to costly problems.

McConell compares building a software to construction work. A building must withstand its own weight at any given point of construction. The same should be true of software. If the modules of software do not play together before every last piece is done, the visibility needed to make progress is lost, the complexity might grow to an overwhelming extent and the product might collapse before it is finished. [17]

We argue that most software will survive until production release, however. Even if the software's quality were less than satisfactory, the investments made into development are too high. The problems in integration cost delays and make maintenance a lot harder and thus more expensive.

Integration is often considered to be a part of testing. This is due to it being done right after the testing phase in development cycle. McConell argues the signifacance of integration that is such that it should be treated as a separate act. [17] Drawing the line between testing and integration is also difficult because unit tests require the separation of modules and functional test require the integration of them. In our context, integration is a separate act from testing as it includes different kind of maintainability problems than testing.

Continuous integration is an idea brought up by Kent Beck as a part of extreme programming (see Section 2.11.4). The idea is that integration is done frequently, every few hours, to see that no changes have broken the integration. As an important part of the continuous integration are tests. These will show immediately if there are problems. [26]

Fowler argues that the integration should be automated. All the steps include compiling, moving files, loading database schemas, or other phases required to make the build need to be done by one script. This way the build process will be faster and less error-prone. Also, tests should be included in the automated script.

When the script is done, everybody in the project can run it on their own machines as often as is required. To make sure that integration is done properly every developer should commit their changes to the mainline of version control at least daily and the changes should be integrated on a separate integration machine. This will reveal configuration problems and other bugs that are not visible in the local systems. [27] This workflow is presented in Figure 8.

Fowler says that the most important part of continuous integration is the feedback that it provides. Thus the build should be fast - at the most ten minutes. This way integration can be done often enough. [27] Beck adds that the builds should last no less than ten minutes, as it offers a break for the developer. [26] We argue that the shorter building times are not a problem, but in fact encourage developers to run the full build script continuously.

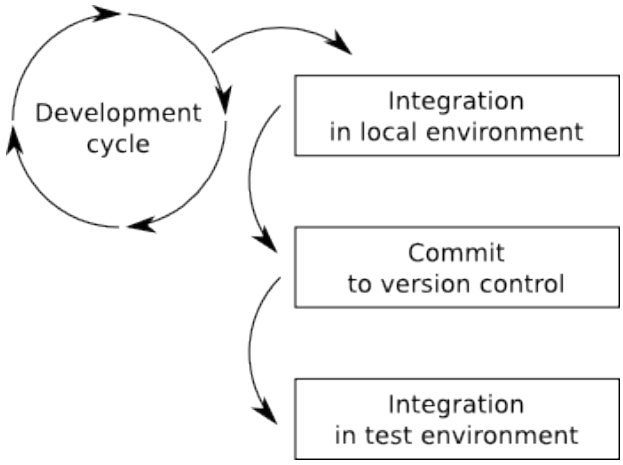

Figure 8: The workflow of continuous integration.

The frequency of "continuous" is not commonly agreed upon. Beck says that integration should be done every couple of hours. This way the feedback is immediate. [26] McConell argues that there is little advantage in doing the integration more than daily. [17]

We observe that the integration should be done in local systems before committing changes to version control, and in the test environment right after it. Commits are usually done after each separate task and thus it is easy to see when integration is broken.

2.9 Testing as a part of maintainability

Software testing is one of the most discussed issues in the industry. There is a lot of literature available on the subject. [28, 29, 30, 31] It is not in the scope of this thesis to cover all the ideas or concepts of testing. We will focus our discussion on testing as a part of maintainability.

Two issues concerning testing are discussed elsewhere in this chapter. These issues are Test-Driven Development in the Section 2.11.5 and Continuous Integration in Section 2.8.2.

2.9.1 Role of testing

- **Testing:** The process of executing a program with the intent of finding errors. [28]

Myers' definition of testing presented above describes the nature of testing. Tests do not exist to show that the software works nor are they written to *prove* that there are no errors in the software. Both are in fact argued to be impossible. [28, 1]

Myers argues that this distinction makes a difference for how tests are written. If a developer is trying to break the software with tests she will thrive to write such tests that will fail. Whereas a developer who is trying to prove that there are no errors in the code will strive for tests that will not fail. It is only failing tests that provide a chance to make errors visible. [28]

In this thesis, testing will be defined according to Myers' definition.

2.9.2 Testing levels

Software testing can be done on several levels. Different levels try to find errors from different parts of software or from different approaches. We present here the levels presented in Burnstein's book Practical Software Testing. They give a proper overview of the kind of tests that are written to ensure that software meets its both functional and performance requirements. [29] This list is not all inclusive and some testing levels might be called with other names. It is, however, sufficient to present the idea of testing levels.

- **Unit testing** deals with the smallest possible testable units in software. What a unit is depends on the program. It might be a function, a procedure or a class.
- **Integration testing** tries to find the errors that occur when different units are combined into groups and full systems. Although the units might work correctly in unit tests, there might be errors in the interfaces or in the way they are used by other units.
- **System testing** evaluates the functionality, quality and performance of the system as a whole. System tests can be divided into following subsets:
  - **Functional testing** overlap greatly with acceptance tests. It is used to test that system fulfills its requirements.
  - **Performance testing** ensures that the system meets its performance requirements and tries to show whether there are some parts of the software that affect performance significantly.
  - **Stress testing** tries to find the stress with which the system will fail as it runs out of resources.
  - **Configuration testing** ensures that application changes according to changes made into configurations.
  - **Security testing** tests the features which protect the data from compromising or stealing.
  - **Recovery testing** causes resource losses to the system and ensures that the software can recover from those losses properly.
- **Acceptance testing** ensures that the product meets its requirements. This is a way for the customers to accept new features.

2.9.3 The importance of tests for maintenance

In the "IEEE Standards for a Software Quality Metrics Methodology", maintainability is defined as consisting of three factors: correctability, expandability and testability. Testability is defined by IEEE as "the effort required to test software". [32]

Here, the ease of writing tests is equated with the ease of making corrective or increasing changes to software. This is not in contrast with the definition of maintainability in this thesis, "the ease of making changes to software" (see Section 2.3.2), as tests are a part of software. The importance of the IEEE definition is that it does bring tests out as a separate part. Due to this separation, the tests have an emphasis greater than, for example, documentation or development processes that are not mentioned as subfactors of maintainability here.

This is not a unique view of the importance of testing. Feathers argues that having no comprehensive tests is the main feature that separate legacy code from non-legacy code. Here, legacy code is not defined by the age of the code, but the quality and maintainability of the code. Feathers suggests that one of the biggest problems in making a change to software is estimating how it will affect other parts of the software. Regression tests exist to tell a developer when a change is breaking other functionalities, thus making changes easier and increasing maintainability. [19]

Martin also discusses testing. He argues that tests and their quality are even more important than the production code itself. This is due to their contribution for such source code metrics as flexibility, reusability and maintainability itself. [21]

It has been discussed that there is a point after which it is not profitable to test the software more. Some argue that investments in the quality are free as they pay themselves back later in the project. On the other hand there are studies that show a decrease in the profitability of improving quality as the project is advancing further. There are also models for estimating how much time is enough to use for testing. [33, 30]

The profitability of testing is debated because it does not increase functionality of software. Although work is done and money spent, no new features are added or business value increased. This is why the testing effort should always be justified.

One's approach to the importance of test coverage can also depend on the methodology used for testing. While developers who write tests after production code test only the parts that they find important, developers who follow test-driven development thrive to only write code that fixes tests. However, even with test-driven development all lines of code cannot always be tested. This subject is discussed further in Section 2.11.5.

2.10 The human factors

Not all causes for bad maintainability are technical. There are factors that are not in the codebase but in the project. Some software development methodologies concentrate on how people work instead of the code they are writing. More discussion on this can be found in Section 2.11.4.

Some might argue that even the technical problems are caused by people. That is why it is important to understand how choosing the right people for the right project affects maintainability.

Kopetz introduces development environment-related factors for maintainability. One of them is the availability of qualified staff. [15] Closely related is the stability of staff. Both imply that even if the product is technically well built, just anybody can not maintain it. Maintainers need to have some knowledge on the technologies as well as the product itself. If the people who maintain the software change frequently, the silent knowledge will be lost. [3]

Brooks describes how communication inside a project depends on the number of programmers. The more developers there are, the more they need to communicate with each other. This often leads to a decrease in productivity if the number of developers in a project is increased. It is also closely related to conceptual integrity, as described in the Section 2.8.1. [10]

The personal interests of developers also affect software's maintainability. Martin describes that it is often schedules, budget [21] or fear of breaking something [31] that lead developers to write bad code. Wang argues that one of the biggest reasons for poor quality is the lack of motivation to write clean code. [34]

Wang has studied software engineers' motivation to refactor their code in a survey. He found that some developers found the motivation without external motivators. They wanted to refactor for such reasons as high self esteem, responsibility of the code and social norms. Other programmers needed to be motivated externally through, for example, threats of punishment or recognition. It was also found that too large effort estimations, caused by bad quality, demotivated the developers from refactoring. [34]

2.11 Software development methodologies

"Software development methodology" refers to a framework in which the development process is carried out. Many methodologies drive for improved maintainability. Two of these methodology families are presented here. They approach the subject from very different angles.

Formal methods are methodologies that use mathematical models for software

development. Formal methods originate from the academic world [35] and have then been taken into use in the industrial world as well. For maintenance, their advantage is in making the system easier to understand and read. [20]

Another family of methodologies discussed here is considerably younger and developed in industrial companies rather than the academic world. They are called agile methods. Where the formal methods improve quality through mathematical structures, agile methods emphasize lightweight processes and documentation. Agile promotes preparing for the change and thus improving maintainability. [36]

Although formal and agile have very different approaches, they are not necessarily opposites to each other. The methodologies can be united to support each other. [37] This thesis will not discuss these combination methodologies further, but they are worth mentioning here.

Both methodology families are presented here shortly. The most popular agile method, eXtreme programming,[38] is introduced to give a perspective on how methodologies may improve maintainability.

2.11.1 Formal methods

Formal methods are methodologies that describe software through notations that have a mathematical base. They can be used for specifying, developing and verifying the system. Most formal methods are built around a notation called formal specification language. Examples of formal methods are the Z notation and Petri Nets. [35]

These methods are widely used in critical systems where the correctness of implementation is necessary. The formality offers tools for ensuring the satisfiability of specification and correctness of implementation. They are also used in documenting software. [35, 20]

Hall writes about the advantage of formal methods for software's maintainability. He argues that having a formal specification helps maintainers to understand what the software is intended to do. [20] As formal methods are standardized and often a visual way of describing logic, they are an intuitive method for both implementation and documentation.

According to Hall, specifications made with formal methods are less error-prone than their non-formal counterparts. It is easier to find the mistakes when requirements have to be expressed in a standardized way. [20] This also means that formal specifications can be tested to some extend. Thus unimplementable ideas can be ruled out earlier.

Unified Modeling Language (UML) is widely used in the industry. It consists of multiple notations that are all their own languages for different purposes. UML's

class diagrams, that describe functionalities and relations of classes, are probably the best known part of UML, but there are many others that can be used as a formal method. For example, state machines are used to show how the application transfers from one state to another. [39]

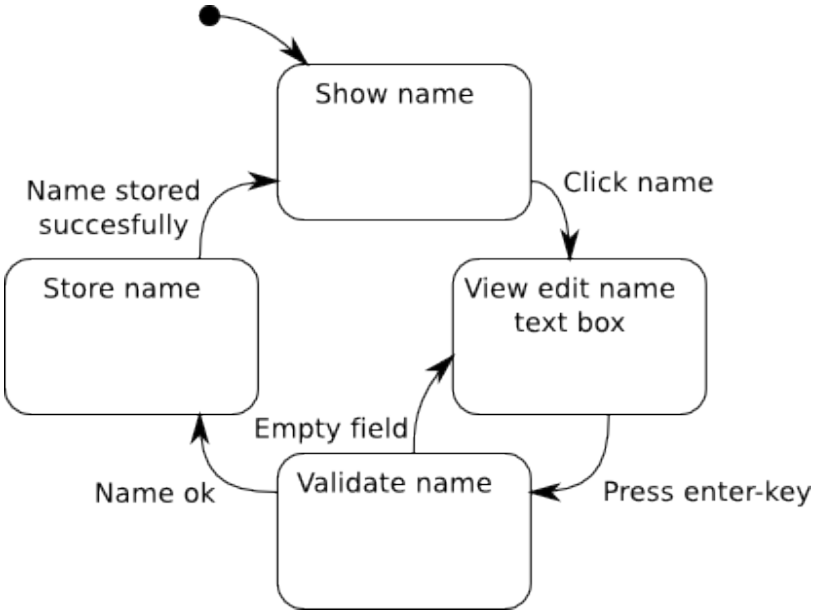

Figure 9: An example of a state machine.

An example of a state machine is presented in Figure 9. The boxes represent states of application and arrows represent transitions. In this example, there is a user interface field that duplicates as both "show" and "edit name field". When the field is clicked, edit-state is enabled. Validation is begun when the user presses the enter key, and if a name is validated, the state returns to showing the name. If the user gives an empty name, he is returned to the editing of name.

Formal methods are sometimes considered controversial, though. It has been argued that they are difficult and suitable for only highly critical systems. On the other hand, some say formal methods will revolutionize the software industry, as they offer a more reliable way of implementing software. Hall claims that formal methods are not as expensive or difficult as some argue, but they alone are also not enough to ensure reliability or correctness. [20] Developers can use formal methods or make mistakes with them, as with any other language.

We argue that formal methods, like any other language, need to be studied before their advantages can be taken into use. A developer with experience from formal methods can use them to reduce errors in software and shorten delivery times. As they have some mathematical background, they might be less desirable for most developers unfamiliar with them.

2.11.2 Agile methods

In the definition of maintainability (see Section 2.3.2), the word "change" has a huge emphasis. Maintainability itself is the ease of change. In the same section, we quoted Yang and Ward, who underlined that the most important part of maintainability is planning for maintainability. Thus the most important part of maintainability is planning for change.

The need for change is expected of a software at some point of its lifecycle. Change is inevitable and uncertain in its direction. [9] Thus it is important to plan the software for a change. This was also an important motivation for creating the agile software development methods. The name "agile" was given by 17 experienced software engineers in 2001. [36] Agile methods is an umbrella term for many practices and methods for software development and project management.

This thesis covers agile methods rather widely, as it was carried out in a company that uses agile methods for daily work. Applications are developed in co-operation with customer companies, and agile methodologies are used to keep feedback loops short during development as well as maintenance phases. Lightweight methodologies fit well in the web and mobile environments, as future requirements are often not visible.

2.11.3 Agile manifesto

Agile methodologies, like eXtreme Programming (XP), Scrum and others, embrace changes rather than trying to avoid them. Agile Software Development Alliance was founded to promote these methodologies. They have a manifesto that suggests which guidelines software development should follow to ease the change.

The manifesto was published online. The whole text is quoted below. [40]

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value in the items on the right, we value the items on the left more.

Fowler and Highsmith, two of the signers of the manifesto, emphasise the word "uncovering". The signers do not claim to have all the answers, but provide a point of view with the manifesto. Their goal is to improve software quality and customer satisfaction. They introduce the twelve principles behind the agile manifesto. [36]

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals, give them the environment and support they need and trust them to get the job done.
- The most efficient and effective method of conveying information with and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done - is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

These principles describe ideas on how to make software development more agile, more suitable for changing requirements. The principles are more focused on the project than on the source code. It does promote simplicity and attention to technical excellence, but does not suggest any definitions for these. Therefore, agile manifesto on its own is not enough to be a framework for software development. However, many frameworks have been developed from these principles, with varying emphasizes. Where eXtreme programming offers an ideological background and a list of practices to be used, Scrum merely discusses how projects should be organized.

We see that the word "agile" is easily misunderstood. Despite the name, agility requires some discipline for following the instructions and practices. Being agile does not mean that changes can be brought into a project at any given moment or that anything about the project can be changed. This is forgotten in many projects where the vendor's and the customer's vision of agility differs.

Boehm has also seen the overresponding problematic. He takes the US Federal Administration's air traffic system as an example for a project that overrun

its budget by $3 billion dollars for overresponding. He claims that both agile and plan-driven approaches have their advantages and pitfalls. Where agility may cause costly mistakes in basic architecture, plan-driven specifications might become obsolete in a sudden manner. [41] It seems that agile methods are best suited for projects that have little visibility for future requirements. With them, decisions can be made along with development, when the the requirements become clearer. Plan-driven methodologies, on the other hand, should be used when the scope of the project is well-known.

In the following section, we take a look at eXtreme programming which is claimed to be the most popular agile methodology. [38] After that we introduce a practice of test-driven development in Section 2.11.5. It is closely related to eXtreme programming as well as other agile methodologies.

2.11.4 eXtreme Programming

As an example of the agile methodologies eXtreme programming, developed by Kent Beck [38], is presented here. Intuitively, a methodology for agility cannot be strict. Hence eXtreme programming is not a checklist nor an exact guide. It is a collection of values, principles, and practices. In this section we will discuss the subject on the bases of Beck's book "Extreme Programming Explained." [26]

Values. Beck describes values as vague top-level ideas that steer people. They alone are not precise enough to guide actions. The purpose of values is to set grounds for the project's goals. Beck says that for software development, it is not important which exact values one uses, as long as the whole team can share common values. He does propose five values for extreme programming, however.

- **Communication** is an important part of effective cooperation, but not enough alone to create a successful software project. Even though communication does not produce functionality it can solve or even prevent problems before they occur.
- **Simplicity** is the opposite of complexity, and it is presented numerous times in Section 2 of this thesis. Simplicity does not mean a team should aim for simple systems. It means that one should always aim for the simplest solution to solve a particular problem.
- **Feedback** is an important part of from Lehman's laws. [11] It offers information on the required changes. An eXtreme programming team gathers feedback as fast as possible to be able to adapt to new requirements quickly.
- **Courage** should not be used as the primary value, as it might lead to recklessness. Beck argues that programmers often feel fear in their daily work. The important thing is how they deal with the fear. He encourages developers to bring up the problems they know exist.

- **Respect** ties all the other four values together. According to Beck, if the people in the team do not respect each other and if they do not care about the product, the result will be failure.

Principles. Beck describes principles as a bridge between values an practices. Where the values are bases of the methodology, principles guide the practices. In this thesis, we will not discuss all the principles separately but merely list them here. The principles are *humanity*, *economics*, *mutual benefit*, *self-similarity*, *improvement*, *diversity*, *reflection*, *flow*, *opportunity*, *redundancy*, *failure*, *quality*, *baby steps* and *accepted responsibility*.

Practices. As the last piece of the chain, practices require principles and values to work. Beck argues that if a development team has a practice without a value, it will become a burden. Practices need to be voluntary and reflect the values, then they help programmers to get most out of themselves.

Beck divides the practices of eXtreme programming into two groups: *primary* and *corollary* practices. We will only discuss the primary practices.

- Beck encourages teams to **sit together**. A common room or space for the developers makes communication easier. It also promotes the humanity principle as people working together have more of a sense of common accomplishment. In multi-vendor projects, this practice might be difficult to follow. Possible problems for multi-vendor projects are further discussed in chapter 5.

- The people in the development team should have the feeling of **whole team**. The feeling of "we are in this together and support each other" grows when the team includes all the competences needed for the project to succeed.

- A team needs an **informative workplace** that gives an idea of how the project is going in fifteen seconds or so. The progress and the real and potential problems should be visible. Also, the workspace should fulfill such needs as water, snacks and sufficient privacy. In practice, we have noticed that making the progress and tasks visible in the workplace focuses the work and gives the team the feeling of control and confidence.

- There are only so many hours a developer can do productive work per day. The practice of **energized work** encourages people to work eight-hour days, because when a developer is tired, it is easy to remove value from product, but hard to see the loss of it. Therefore, one should get enough rest and provide sufficient rest for the rest of team also. In many cases, we have noticed that longer days have led to increasing technical debt. Although it is tempting to do more than eight hours a day, it should be avoided whenever possible.

- **Pair programming** might conflict with the requirement of sufficient privacy. Beck argues that with mutual respect and consideration it can be made to work. The advantages of pair programming include improved communication, more focused work and more disciplined coding style. We argue that the possibility for pair programming depends on the development team. Working in such intimacy might be inappropriate or uncomfortable for some cultures.

- Beck proposes that software projects should not have requirements as most features are not mandatory for the release. Instead, teams should talk about **stories** that are customer-visible. Stories make the estimation of cost and necessity easier if their cost is estimated in an early phase. Stories are used in everyday work even outside the eXtreme Programming context. They are suitable for application with wide user interfaces. Many industrial applications might still need sufficient requirements.

- The **weekly cycle** starts with planning the stories to be done during the next five days and splitting the stories into tasks. Then, automated tests for the stories are written and the rest of the week is spent making those tests pass. This practice is widely in use, as weekly meetings are nothing new for office environments.

- Beck says that a **quarterly cycle** is a long enough period for developers to plan the project without concentrating too much on how the current tasks affect the future. In quarterly meetings, the team should pick a theme for the next quarter and enough stories to follow that theme. Quarterly cycles are suitable for long-running projects. For many web or mobile applications, cycles should be shorter than that.

- **Ten-minute build** is long enough for developers to drink a cup of coffee but not too long for them to get irritated for waiting. Beck argues that it is important to have an automated build which builds the whole system and runs all the tests. And that the build should not take more than ten minutes. Automation ensures that all the building blocks are in place and there is little room for human error in the process. As the results show, builds longer than ten minutes encourage teams to run them without tests. This increases the possibility for broken releases.

- **Continuous integration** is discussed in Section 2.8.2.

- **Test-driven development** is discussed in Section 2.11.5.

- The process of **incremental design** means that you should design the system every day so that it best describes the picture of the product that particular day. All the decisions should not be made at the beginning of the project but the stories should be brought in incrementally at the last *responsible* moment possible.

The values and practices presented in this section are not the whole story of eXtreme programming but they give a picture of the ideas behind it. The goal is to embrace the change and thus improve software quality and especially maintainability. There is acceptance in the industry that these goals are also met. [38]

Although all of the practices mentioned here are not used in the company we wrote this thesis in, they are mostly recognized as suitable tools for web and mobile development. As many practices concern relations between team members they will not necessarily work in all countries or cultures. For example, making the team work together might require changing team members. In a small company, hiring or firing an employee could be difficult due to Finland's legislation. Also, letting people go might affect others' motivation for their work.

2.11.5 Test driven-development

Testing is an important part of maintainability. We discussed testing more closely in Section 2.9. In this section, we will look more in-depth at a methodology for testing and developing. The methodology is called test-driven development.

It has been argued by many experienced software developers, including Fowler, Beck, and Martin, that quality benefits from writing tests before production code. [12, 26, 21] This is not necessarily intuitively visible. Many developers consider writing tests somewhat of a burden and too time consuming. Fowler argues that when tests are done in beforehand, debugging time will decrease to almost zero, thus in fact making development faster. [12]

Fowler's argument on debugging is based on his experiences, and might vary from an application or a development team to another. However, the result seems trustworthy. Disciplined test-driven development guides the developer to write only necessary code, thus making the implementation simpler. Also, minor mistakes are instantly brought visible, as there is already a test to raise a warning before the mistakes are even made.

Martin introduces the three laws of Test-Driven Development that explain the procedure of this methodology.

- You may not write production code unless you have first written a failing unit test.
- You may not write more of a unit test than is sufficient to fail.
- You may not write more production code than is sufficient to make the failing unit test pass.

The aim of this procedure is to create a development loop. First, a short test that fails is written, and then a short piece of production code is done to fix it. This is illustrated in Figure 10. The cycle should take just a few minutes. Of course, it is not

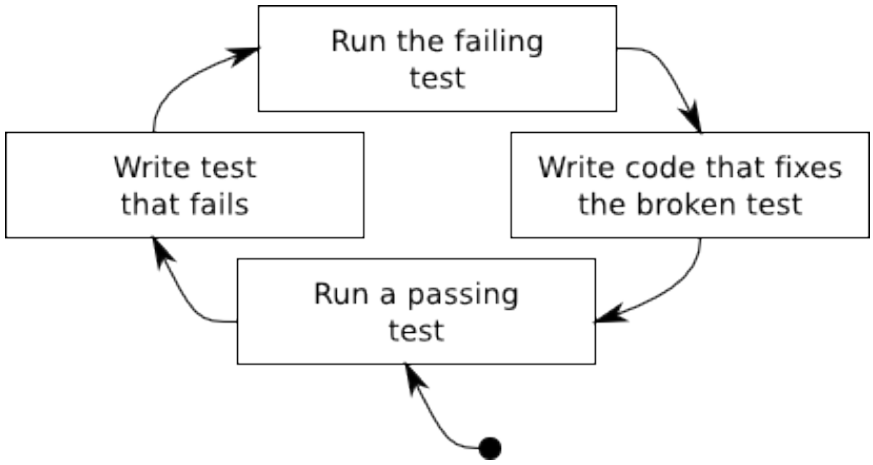

Figure 10: The development cycle of Test-Driven Development.

always possible to write tests in such a procedure, thus this is merely a guideline. [31]

Both Martin and Beck argue that Test-Driven Development keeps developers more focused. It is easy to get lost for hours if one does not have a clear procedure. They both also write about more trusted code that enables not just the author but also others to change the code when more confidence is needed. [31, 26]

There are cases where the tests-first approach is not easy to use. A developer might be restricted by a third-party library, legacy code or, for example, an embedded environment which prevents this methodology. Martin argues that Test-Driven is as much of an attitude as it is a discipline. Thus impediments that make testing more difficult should not prevent testing altogether. [31]

As tests themselves do not implement any functionality, Test-Driven Development should be considered only as a tool that can be used. We have seen in our daily work that many people believe that this methodology would improve quality, but are too unsure to take it in use. Changing the way code is written is not easy. Therefore, trying Test-Driven is often postponed, as there is little time for learning new workflows in an average project.

3 Research questions: How to improve maintainability?

- How can maintainability of application software be improved?
  - How can we estimate maintainability?
  - Which factors affect maintainability?
  - What instructions to give to development teams?

Estimating maintainability in a quantitative manner is difficult and there is no commonly accepted measure for it. There are, however, many factors that are agreed upon to affect maintainability. Most of those factors can be estimated or measured with statistical computations.

As the title of this thesis states, we are not aiming to find a measurement of maintainability. We aim to improve maintainability. Thus the actual amount of technical debt or other maintainability factors is not in essence. Instead, it is important to understand which practical steps can be taken to ease the change.

It was found in the literature review that maintainability is hard to build into software that is already in the production phase. Maintainability should thus be considered from the very beginning of any software project. This is why we focus on finding ways to improve maintainability before the product is in the maintenance phase of its lifecycle.

Focus, in this thesis, is on practical actions that development team can take. With this specification, we can divide the main research question into three sub-questions, each supporting one another.

How can we estimate maintainability? As is mentioned many times estimation of maintainability is not a trivial task. Rather than comparing the various statistical approaches introduced by others, we focus on what can be done during a project to better understand the underlying technical debt.

Which factors affect maintainability? Some might argue that maintainability factors ought to be known before they can be estimated. Although this is true to some extent, we will first review the code base and only after that evaluate which factors seem to be the most meaningful. We will also discuss the profitability of maintenance improvements, even though a proper coverage for this subject would require a more thorough study on the financial impact of bad maintainability.

What instructions to give to development teams? It is important to summarize the results from the first two questions to a set of instructions and recommendations. This is the task of the last sub question. With it, we aim to provide an empirical list of practical steps for improving maintainability.

4 Research methods: The project audit process

To answer the research questions three project audits were made. In the literature review, it was said that there have been two kinds of estimators for maintainability. Some have been based on expert reviews and others on code analysis, also surveys had been done to estimate motivation for refactoring.

The project audit process is somewhat of a combination of these three methods. Code analysis tools are used to guide the expert review. No actual survey was done, but the development team was included in the process through discussions.

There could have been other ways to get more statistical results for maintainability evaluations. But as was concluded in the literature review, those do not necessarily describe the ease of actual maintenance work. Thus a more practical approach was adopted.

This section introduces the audit process which was used. Before the process is described its background and goals are introduced. Finally, a more thorough evaluation of the chosen method is given in Section 4.1.4.

4.1 Case study: Project audits

A case study of a project audit process was done during the thesis work. The audit process was designed as a part of concept design for the Lifecycle Management team. The goals for the audit were:

- To transfer knowledge from the development team to Lifecycle Management.
- To work together with the development teams to estimate the maintainability and the amount of technical debt of a software in the development phase.
- To suggest improvements that will decrease the risk factors for maintainability.

4.1.1 Background

Futurice is a middle-sized software company founded in Finland, with locations in Helsinki, Tampere, Berlin and London. Futurice produces web and mobile applications for its customers. In the spring of 2010, Futurice started internal development for improving maintenance. For this task a new team called Lifecycle Management was founded.

The new team was to responded to several challenges. The main objective was to make maintenance work more interesting for the maintainers, offer better quality for customers and improve the profitability of maintenance for the company. Earlier maintenance had been done in an ad hoc manner, which left room for improvements in all three aspects.

The concept was to create a maintenance team that would not just inherit projects as they were released, but also participate in projects already during the development phase. This would ease the knowledge transfer and give the maintainers a possibility to affect the quality and maintainability of the projects before they were moved to the maintenance phase. The responsibility of the team would thus stretch throughout the project's lifecycle and not just the maintenance phase.

A part of the concept was that Lifecycle Management could decline the transfer of a project from the development team to maintenance if quality standards were not met. This was seen as a crucial part of assuring proper possibilities for fluent maintenance work as well as of tying the financial risk of the maintenance phase to development. As project participation was not possible for all projects, another way of learning the projects and estimating their technical debt was needed.

To answer this requirement an audit process was designed. In this process a member of the Lifecycle Management team would get to know the project with the development team as early in the development phase as possible. While learning the project the maintainer would also give suggestions and instructions on how to improve maintainability and thus ease the transfer of the project to maintenance later on.

The audit process were designed to be lightweight and coaching instead of heavy and judging. The target was to aid development teams with improving product maintainability instead of ordering them to fix defects. On the other hand audit process was planned to promote good practices throughout the company. A four-step process was designed to achieve these goals.

4.1.2 Process description

The audit process was designed to consist of four steps that would benefit both maintainers and developers. The first two steps are for knowledge transfer from the team to the maintainers. The third step is for coaching the team and the fourth one is for documenting the results.

1. **Project walkthrough** In the first phase a member of the development team introduces the project to the auditor. The basic features, architecture and most important documentation are presented. This phase was designed to last less than half a working day.

2. **Expert review** After the project is introduced to the auditor, he will do the actual audit on the project. This phase includes setting up development environments and going through the audit checklist, which is presented in Section 4.1.3. The most important part of this phase is for the auditor to get a proper understanding of the overall quality and find the possible risks concerning maintainability.

3. **Team discussion** After the review, a discussion is held with the team. The discussion should not be judging but coaching. The team will first give their own opinions of the quality and maintainability of the product. Then, the auditor can present his suggestions or instructions, if they are not brought up by the team.

4. **Report** As the main goals of the audit were to transfer knowledge and coach the team, the reporting is not of great importance. Some documentation should be written though: one internal report of the findings and possibly another less technical one for the customer.

After the first audits, the audit process was further developed. The improvements included following the progress of improvements that were suggested, as well as keeping more closely in touch with the project team. The improvements are further discussed with the other results in Section 5.1.4.

4.1.3 Checklist

The checklist is a guide for a maintainer who is doing the expert review phase of an audit. It includes many good practices mentioned in the literature review of this thesis, along with some Futurice's internal recommendations.

The checklist was not designed as an exact list for guided expert review (see Section 2.3.2). Its purpose is to be a reminder and aid for the auditor. It consists of the following sections:

- **Good practices** such as proper setup guides and sufficient documentation, proper use of version control and deployment processes.
- **Signs of technical debt** like poor test coverage, broken tests and insufficient logging.
- **Convention abuse** in the code base including inconsistent naming, forming and long functions or classes.
- **Architectural risks** in the system design or selected technologies.

For different projects a different type of set of checklist items is needed. Therefore, during the project introduction phase of each audit, the maintainer needs to define which aspects are required to be checked. The predefined list can be used as a base for this evaluation.

4.1.4 Audit process and maintainability

As the goals of the audit process were not exclusively aimed at evaluating maintainability the chosen method has to be inspected from the viewpoint of this thesis. The audit process deals with a variety of aspects from quality evaluation to promoting

good practices. Maintainability is undoubtedly a part of quality and thus part of the audit process evaluations.

After the audit process was developed, three different projects were audited using it. Experiences from each of these projects restricted with the subjects introduced in Section 2 were used in the results of this thesis. The goal was to find answers to research questions from the audits.

Although the number of audits is small and technologies reviewed are quite similar to web and mobile technologies, we succeed at finding many indications of technical debt that were mentioned in Section 2. We also found that those have caused difficulties in making changes to software. We argue that the audit process is a proper tool for evaluating the research questions.

There would have been other options to investigate maintainability. Surveys have been used by others to find out what developers consider to be bad maintainability. [25, 34] Others have used computed metrics to evaluate maintainability or factors of it. [5, 6] Surveys could describe which factors affect maintainability, but as the restrictions for change are different in the development and maintenance phases, we needed to focus more on how maintainers themselves see maintainability and how they can evaluate an individual application. Statistical metrics, on the other hand, could point out the problems in applications, but their view on maintainability is rather restricted, as they cannot see maintainability problems outside the code base.

The title of this thesis implies that we are focusing on instructive conclusions that offer ideas on how to improve maintainability. Thus we use the experience of professional maintainers and the basis provided by literature to find out how to instruct developers to ease the maintenance phase. This is also one of the main goals of the audit process.

5 Results: Answers to the research questions

In this chapter, we answer the three research questions set in Section 3: how can we estimate maintainability, which factors affect it and what instructions can be given to development teams. We discuss the factors mentioned in the literature review and reflect the audit results to these factors. Many of those factors are visible in the project audits, but some new aspects also arise.

In Section 5.1, we argue that developers are usually well aware of the maintainability of their product. We discuss how to find this silent knowledge and help transferring it to the maintenance team. We also take a look at the audit process and discuss which ideas worked and which did not.

Section 5.2 includes the issues that were brought up by the auditors. This section argues that neither code metrics nor expert review alone can give a precise enough picture of maintainability in any project. Both are needed and some of the important issues might still be missed.

5.1 The audit process: How to estimate maintainability?

To estimate maintainability, an auditor needs to know the project well enough to understand the underlying problems. Different techniques, tools and measurements have been suggested. [4, 5, 6] To find maintainability issues of a project in its development phase, we used an audit process. In this section, we discuss how well the audit fits the task, what were the good sides and what could have been improved.

5.1.1 Discovering silent knowledge

One of the most clear results of the audits was that developers are well aware of technical debt in their software. They know the points where the code is smelly and which modules are hard to change. Developers have a realistic picture of the overall quality, and they are honest about it.

Developers are also rather well aware of good practices of coding. Expertise grows along with experience, and thus especially the senior staff is capable of evaluating the amount of technical debt in their projects. This result is rather intuitive; those who have made the design decisions are aware of the implications as they should.

Even though developers have the knowledge the amount of technical debt seems to stack in projects. As we mentioned in Section 2.10 about human factors of maintainability, there are many reasons why developers leave technical debt untouched. Many of the same factors that are mentioned by, for example, Martin and Wang, were acknowledged also in audits. This is discussed further in Section 5.2.1.

This silent knowledge is crucial for the audit process. It was quickly recognized that this knowledge is easily left unused or forgotten when people in the project change. One of the biggest problems is thus transferring the experience of the development team to maintainers. The next two subsections discuss the ways in which the knowledge was transferred, what worked and what did not.

5.1.2 Learning the project

This section describes the lessons learned in the audit process through four process steps. In each step, we will discuss the good and bad sides and how we managed to learn about maintainability. An exercise done in the fourth step is further described in Section 5.1.3.

Project walkthrough. The first phase was planned to be a quick introduction to the project. It was done with a developer from the audited project. The most important thing in this phase was that the auditor got an idea of what the project was about and where source code and documentation could be found.

Project walkthrough brought up the biggest problems concerning the structure of software and integrations with other software. It was learned that it is not worthwhile to spend a lot of time in this phase. Having the knowledge of technical debt is not enough, but the auditor must also be able to understand the structure of application better to have a meaningful discussion about it.

Lessons learned from project walkthrough:

- The auditor needs to have a general understanding of the product before in-depth discussions with developers can be had.

Expert review. The second phase was planned to last from one to two days. This was found to be too short a time. As a modern software often is large and complex, learning even the basics quickly seemed to be impossible. In practice, this phase took from five to seven days.

As is explained in the literature review (Section 2), there are two principal ways for maintainability evaluation: computing code metrics and expert review. In the audit process, a combination of these two was used.

No official metric numbers were done or given to the development team. Instead, code analysis tools were used to find signs of technical debt. There are several tools available for different languages. For audits, we used Roodi[1], CodeNarc[2] and JSLint[3]. They are mostly meant for features that are easy to compute, like following

[1] https://github.com/martinjandrews/roodi
[2] http://codenarc.sourceforge.net/
[3] http://www.jslint.com/

certain code conventions, but some of them can also compute more advanced features like cyclomatic complexity (see Section 2.6.3).

Expert review was done by reading some examples of code files and trying out some minor changes to the code base. A very important part of this phase was setting up the development environment. This usually brings up many problems that are hidden from automatic tools. Problems with integration can only be seen when the application is running. Also, a better view of logging and documentation can be gained when the application needs to be installed to a local environment. More on the importance of these factors can be found in Section 5.2.

The auditor was also added to the development teams' internal communication channels. This way, he could follow the discussions and pinpoint some problems that were not found in the review otherwise. If some module or certain procedure, deployment for example, was discussed often, it was a clear sign that there were some problems in that area. These problems were then discussed in the third phase of the audit.

Lessons learned from the expert audit:

- Code analysis and test coverage tools can pinpoint problematic modules.
- The auditor easily gets lost in details when reading too many source code files. It is not an efficient way to investigate code related issues.
- Including the auditor in the development team's internal discussions brings up silent knowledge that the auditor did not know to specifically ask.

Team discussion. The third phase was to have a meeting with the whole team to conclude their thoughts after review. The auditor's role in this discussion was more instructive than judging; the question is how the team itself sees the maintainability. This is done so that maintenance can learn more silent knowledge. At the very end of the discussion, the auditor gives tips and possible requirements for improving maintainability. It is important to note here that fulfilling these requirements also needs to be followed. This is discussed further in Section 5.1.4, where improvements for the audit process are suggested.

As many maintainability factors are hard to find with metrics or reviews the silent knowledge is in an important role in the audit. An exercise was planned to steer the discussion, so that the application will be covered as widely as possible. The exercise was called "fake planning poker" and is described in Section 5.1.3.

Discussions were found out to be useful. Unlike in project walkthrough, the auditor had a rather good picture of the project at this point and could focus on the problematic points. He could also clarify details that were unclear during review.

Lessons learned from team discussion:

- Developers are well aware of the maintainability problems. Their knowledge is sufficient, but they need to be motivated and given enough resources in the budget to refactor the code.
- Clear guidance on how to improve maintainability must be given at this point.

Report. The last phase is basically just taking notes. A full review report was written to the company wiki. As this format is too heavy for efficient knowledge transfer, a slideshow of the conclusions was done.

The reporting part was, as planned, very lightweight and did not add any new information. The lessons learnt in reporting were:

- The report should be done as a slideshow that only includes the most important conclusions.

5.1.3 Using fake planning poker

Background. Many aspects of the code are hard to see in a review. Dead code, unexpected dependencies, and many other features were invisible in reviews. As explained in Section 5.1.1, the team is mostly aware of these defects but they do not easily come up in conversations.

During the audit process, an exercise was developed for team discussions. The goal of this exercise was to aid the team to think through different parts of the application and try to explain the difficulties that might come up during changes. The exercise was based on planning poker.

In agile development, requirements are divided in user stories. These are usually a few sentence long descriptions of features to be implemented. In planning poker, work effort for the stories is estimated with the whole team.

First, an individual story is discussed so that everyone has an understanding of the requirements. Then, each developer independently decides on a work estimate, and the estimates are revealed at the same time. The developers with the highest and the lowest estimates justify their opinions. Through this discussion a unified estimate is given. The name for this method comes from revealing one's hand in poker. As poker players show their cards simultaneously, so do the developers reveal their estimates - often presented with numbered cards. [42]

Playing the game. In the audit process, the idea of planning poker was used to discuss the potential maintainability risks. A series of user stories was planned with no correlation to the real project. The aim was to cover the application as

widely as possible with the stories thus making the developers think through the implementation.

In team discussion, the auditor presented the imaginary user stories and answered developer questions in a kind of customer position. In some cases, discussion of the story brought up new ideas, and the story was adjusted to bring more details in the game.

After the team had revealed their estimates, a wider discussion on their opinions was held as in real planning poker. In this discussion, the auditor used his knowledge on the particular project to steer the discussion and asked more precise questions on the motives of the estimates.

Experiences. As this exercise was not repeated systematically, nor was it measured in a precise way, we can only describe the experiences and not propose its usefulness in wider cases. We can, however, present some conclusive thoughts on the method and how it could be used in the future.

During the discussions, fake planning poker succeeded in bringing out some silent knowledge. Particularly, invisible problems in the application's structure were clear after they were explained as user stories. Also, some problems concerning project organisation and co-operation with other vendors came up in discussions.

Planning poker has some obvious problems. As with real planning poker, the estimates are dominated by the stronger personalities in group. [42] Also, some major problems might be missed as all the modules of the software are not covered with the example stories.

Regardless of its problems, fake planning poker can be a useful tool for bringing up issues in team discussions. It provides a better structure and an opportunity for each member of the team to participate.

5.1.4 Ideas for improving audits

The audit process was found to have two major difficulties. The first one is making the underlying technical debt visible to both for the auditor and the team. The second one is how transferring the silent knowledge from team to auditor.

Visualize technical debt. As technical debt is left in code base, it is gradually forgotten. We found that even though development teams have an understanding of the technical debt, they often cannot name problematic sections precisely. Making technical debt visible does not only guide the auditor straight to correct components, but also reminds the team of the work left undone.

Making technical debt visible is not a straightforward task. As was explained in Section 2, no metric can cover all the possible signs of technical debt. However, code metrics can help a great deal on understanding what kind of debt is found from the code base.

Many code analysing tools were found useful in the audit process. They search for convention violations from the code, thus they could also be used to gather statistical data on the technical debt. This statistical data could then be used for visualization.

In Figure 11, an example of a histogram is presented. It presents the number of naming convention violations recognised from 207 source code files. As can be seen from the figure, most files have very few convention violations. The files that have violations have a rather large amount of them. The histogram could thus be used to indicate that there are files that need refactoring for naming.

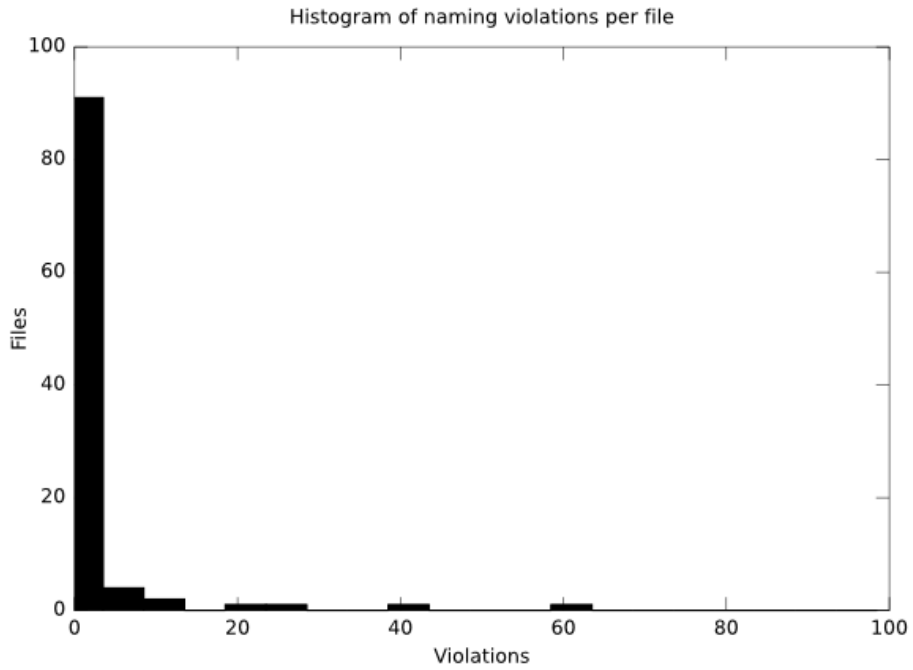

Figure 11: An example histogram of naming violations in files.

Another example that could be used is Table 2, the list of the most common errors. It is presented in a table rather than a visualisation, but it can also quickly show that the most common violations are too short and long variable names. Having underscores in variable names is close but all other violations are not that common.

The examples presented here are not from audit projects. They are instead computed from Log4J[4] open source project. A code analyser, called PMD[5], was used

[4]http://logging.apache.org/log4j/
[5]http://pmd.sourceforge.net/

Table 2: An example list of the most common naming violations

Avoid too short variable name	460
Avoid too long variable name	136
Variables that are not final should not contain underscores	128
A getX() method which returns a boolean should be named isX()	26
Abstract classes should be named AbstractXXX	15
Variables should start with a lowercase character	9
The field name indicates a constant but its modifiers do not	9
Variables that are final and static should be in all caps.	6
Field name with the same name as a method	2
Method name and parameter number are suspiciously close to equals	1
Field name matching the declaring class name	1

to find the errors. Note that conventions in PMD default naming ruleset might not be the same that the Log4J development team uses. This is not important though, as these are only examples of possible uses of code analysis and not examples of good or bad maintainability.

Follow the progress. Another improvement proposed to the audit process was to follow the progress in the project. As it is rather easy to keep in touch with other teams in one company, we found it to be useful that the auditor stayed on the email lists and other communication channels of development team. This way, knowledge gathered in the audit was not forgotten easily. It is important for the auditor, not to follow every conversation in detail, but to understand the general state and direction of the project.

Another important thing was to follow how development teams would fulfill the improvements to maintainability that were proposed during the audit. It is easy to forget tasks when customers keep pushing stories to the backlog of the project. Therefore, it was important to check every now and then whether the tasks had been done.

5.2 Findings: Which factors affect maintainability?

The literature on maintainability concentrates on defining technical factors. The experiences of the maintenance team in Futurice, as well as the audit process carried out in this thesis, support this approach but also show that other factors are important as well. Logging, documentation, version control, and even sales affect the ease of making changes.

In this subsection, we discuss which maintainability factors are required in each step of making changes to the software. The steps are loosely based on an algorithm for making a change, as proposed by Feathers. [18] We have added project management and deploying, which are not strictly a part of making the change, but affect

the maintainability.

5.2.1 In project management

The availability of staff. Kopetz lists the availability of qualified staff as a maintainability factor. [15] In our project audit, this issue received attention. As project teams can quite freely choose their own technologies, maintenance has to resource the projects as they are transferred to the maintenance phase.

In one of the audited projects, the development team had chosen multiple technologies that no one in the maintenance team knew well enough to maintain. This led the maintenance team to face with three options. The most obvious solution is to hire someone capable of maintaining the project. There are several problems to this intuitive solution. Qualified programmers are rather expensive, and they are not easy to find for any specific technology. In this particular case, there were several technologies that needed to be handled.

Another option is internal transfer from the development team to the maintenance team. This would not only solve the technology problem but also bring silent knowledge of the project straight to maintenance. The problem with this solution is that someone in the development team should be interested in working in maintenance. Also, the development team management has to agree that the developer is free to transfer from one team to another.

The third option for closing the expertise gap is training someone from the maintenance team. In this case, two things are required. Firstly, there has to be someone motivated to learn the technologies, and secondly, there needs to be budget for training. Here the development team and the maintenance team face the question of who is financially responsible for the training.

No easy solution can be found but two important guidelines can be proposed. Neither solve the problem of resourcing directly, but both ease the work indirectly through the technology selection.

- *Use of company-wide standardized technologies* allows maintenance to resource the project easier with qualified staff. As most project are built with only few technologies, the same people can maintain multiple projects.
- *Including the maintenance team in the project design* is another way of easing the transfer. Maintenance can either influence the technologies used, or accept the use of non-standardized technology and have enough time to prepare for it.

At Futurice, a list of accepted technologies was made based on the maintenance team's competences and interests. The list included languages, frameworks and tools that were promoted by maintenance. Other technologies were not disallowed

but development teams were instructed to discuss their decisions with maintenance before choosing a language that is not standardized company-wide.

Definition of done The definition of done (DoD) is a list of requirements that a feature needs to fulfill before it is accepted to be done. Examples of these requirements could be unit tests written, acceptance tests written, all tests pass, code review done, or documentation review done. The purpose of DoD is to reduce technical debt.

If "done" is not defined properly, it is easy to leave quality work for later and then forget about it. As mentioned in Section 2.4, this means that the project's technical debt has grown. Also, Wang proposes that social norm, the willingness to follow common rules, is one of the key motivations for developers to refactor their code. [34] Definition of "done" sets a formal norm for the software's quality.

In audits, DoD was recognized as one of the key elements in following code conventions. If the team did not have commonly agreed upon formal definition of "done", all developers could use their own preferences. The importance of code conventions is discussed in Section 2.6.1. Some teams had agreed on "done" informally. We saw that this already gave the team better confidence on their own quality.

Audits also showed that the definition of "done" should not be too vague. If the definition cannot be followed literally, it will not be used. Instead, it will only remain as a part of documentation.

In short, our findings on the definition of "done":

- Development teams should define "done" *formally at an early stage* of the project.
- The definition of "done" should include *strict technical quality requirements* to prevent technical debt.

Backlog. Backlog refers to a list of stories that are requested by the customer. How backlog is used depends on development methodology. For example in eXtreme programming backlog stories are chosen in priority order for implementation. [26]

The audit process showed that if backlog was missing or was not used in an organized way, the project easily lost focus. Some tasks were replaced half-way through with another task, while others were made in advance just in case they would be needed in the future. Both of these practices increased the amount of dead code in the project. Dead code has been proposed as a bad smell by Mäntylä and others[25], and is discussed in Section 2.7.2.

- Tasks should be *prioritised and handled* in an organized way to avoid lost of focus and dead code.

Multiple vendors. Many software systems today are implemented by more than one company. This turned out to be problematic for maintainability, if communication between vendors was not fluent. Our audit projects included one to three different vendors.

Changes in interfaces often require simultaneous updates to two or more software products. This requires careful planning and continuous communications between the vendors, as well as keeping the customer informed. Bad communications caused delays in our audit projects and thus uncertainty for estimations.

Another problem that arose from the audit was that if components from different vendors share resources, it increases the uncertainty in the project. The common resource might be a server machine or a database. Although resources in our audit projects are not as critical as in embedded systems, they still require excessive amounts of communication between vendors.

The multivendor project factors for maintainability are:

- *Continuous communication* between vendors creates more stability and confidence in changes in interfaces.
- Different systems should use *resources through APIs* instead of sharing them. Isolating components from each other lessens the coupling of modules, making structure of software more understandable.

Financial realities. In many cases, the quality is affected by tight budgets or schedules. As was discussed along with the definition of technical debt in Section 2.4, leaving quality improvements undone during development does not necessarily bring savings. In the project audits, it was observed that this holds true also in practice. Complex structures and unclean code has an effect on the effort required for changes. Bad maintainability increases costs in the maintenance phase.

The problematic relationship between budget and quality is enforced by the fact that initial development and maintenance are often done by different teams from different budgets. Sometimes even the customer changes when the product is transferred to maintenance. People who are responsible for the profitability of the development phase have no responsibility on the profitability during the whole lifetime of the software. This leads to a situation where it is reasonable for the manager of the development team to cut costs from the quality effort.

The reasoning behind reduced quality work is not completely wrong either. Some refactoring efforts are more meaningful than others. For example, it probably pays off to reduce coupling between continuously changed modules, but making small improvements here and there in rarely changed modules might cost more than it produces later.

Notes on financial realities:

- Making room in the *budget for refactoring is essential* for keeping the product maintainable.
- The *refactoring effort has to be directed* to the most important maintainability factors.

5.2.2 When identifying change points

Identifying the change points is the first step in making the actual change. After a request for new feature or an error report is given to the developer, she must be able to pinpoint where the changes need to be done. Proper documentation and logging as well as clear structure are important for understanding the change.

The structure of the application is also very important for breaking dependencies. It is discussed more in Section 5.2.3, where we will discuss the software's internal structure. In this section, we will concentrate on problems on the system level.

Sufficient logging. For corrective changes, logging is essential. When an error report is received, it is important to be able to find out the conditions where the error occurred as exactly as possible. [17] This way the error can hopefully be reproduced.

In the project audits, it was found that logging is problematic for many developers. One should only log the most essential information, because logging can be costly for resources. On the other hand, too little logging is not enough to describe the conditions.

Usually, logging levels are used to avoid excessive logging. Each log message is set with a level that marks its importance and type. For example, a commonly used logging library for Java, log4j[6], has the following levels: OFF, FATAL, ERROR, WARN, INFO, DEBUG and ALL. OFF and ALL are only used by configuration, but other levels can be set to individual messages.

With levels, different environments can be configured to show less or more logging. Where the local development environment might show all logging production might only show logging messages from WARN and up. Also it was found useful in audits if logging levels can be set separately in the different components of the systems. This way, only the logs that need to be viewed are printed and resources are not wasted on excessive logging.

Another problem for logging is that some environments are unreachable for reading logs. Although in our audits the problematic parts were mobile or web clients, the same problem is present in, for example, embedded systems. In one audited web

[6] http://logging.apache.org/log4j/

application, this was solved by making a separate interface that asynchronously sent logging from the browser to the server. This way the project ensured that important information is available for developers if needed.

Findings of logging in audits:

- The application should use sufficient logging *with appropriate logging levels* and *enough details* of conditions.
- If the application's environment is not reachable by developers the possibility for *logging over the Internet* should be considered.

Thorough health check. Health check refers to a process or tool that shows whether application is running correctly. Usually, health checks are run automatically. If the check fails, an alert is sent to maintenance.

Health checks give maintenance a possibility to respond faster in error situations. They might also include logging which helps to identify where the error has happened. Health check should be able to confirm that the whole system is running correctly. This might mean confirming e.g. server status, databases, and integration connections.

Health check findings:

- Thorough health check for the system *enables early detection* of problems for maintenance.

Guiding documentation. The Agile manifesto values working software over documentation. [36] This is understandable, as documentation easily becomes a burden that is very hard to maintain. [10] Documentation suffers from the same problematic nature as comments. As much as they might be useful for understanding the code, they may also be misleading with deprecated information.

The difference between comments and documentation is that documentation is mandatory to some extent and cannot be replaced with code conventions. Documentation should be a guide that tells the developer where to find tools, how things are done in this project, and how the program itself is constructed. Even if documentation would not cover all the details in an exact way, it should be able to point in the right direction when there is a problem.

Balance between guiding documentation and maintenance burden is fragile. Some features might need more coverage than others, but documentation should not be the code rewritten in another language. Rather it should explain the meaning of features and design decisions. Figures and illustrations, like UML diagrams of the

classes, are usually helpful for understanding the structure. More high-level architecture should also be visualized.

When making a visualization, a standardized diagram might be helpful. Languages like UML offer ready-made standards that are widely understood. But it is not necessary to use any particular language or illustrations, as long as the figure is clear.

An example of such architectural illustration is presented in figure 12. It includes all high-level components, technologies, and directions of data. Other useful illustrations include, for example, user interface wireframes, integrations to other systems and workflow diagrams of complex business logic.

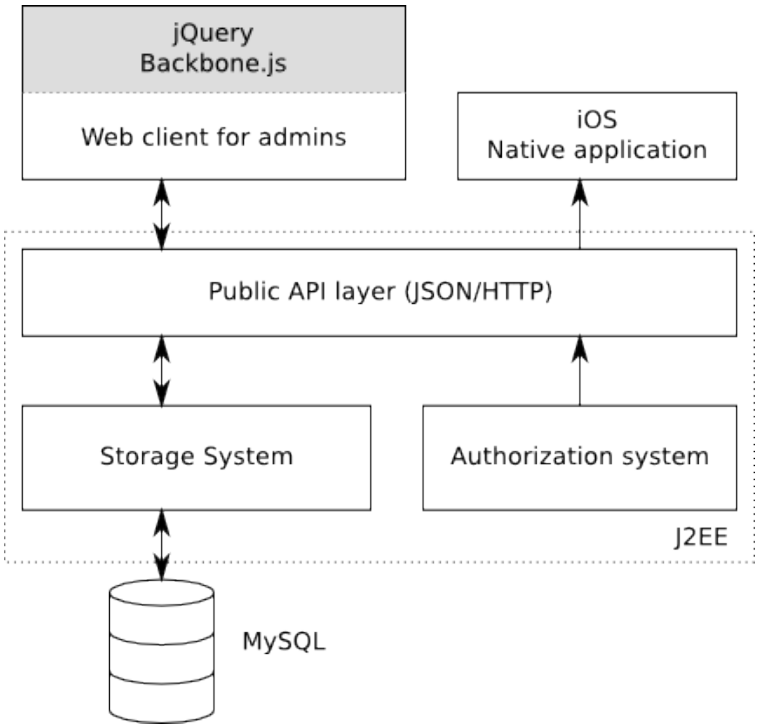

Figure 12: An example of an imaginary architecture.

Also it was found that if software libraries do not have proper API documentations, it decreased the maintainability of applications that use the libraries. Even if the original development team had been familiar with the particular library, the maintenance team might not have any experience with it. This would make it very hard to understand how the library is used.

Kopetz lists standardized documentation as one of maintainability factors. [1] This was seen in audits, but not as one of the biggest problems. If the documentation is in an intuitive place and contains a sufficient amount of information, it was considered to be enough.

Key notes on documentation by auditors:

- Documentation should answer at least the following questions:
  - What is the *high-level architecture* of the software?
  - Where is the *source code* stored?
  - How to *set up the development environment*?
  - Which *tools, procedures and practices* to use in this project?
- *Illustrations* are often useful when documenting complex systems or logics.

Access and security of the system. We found in audits that one of the most critical factors of maintainability in multiple vendor projects is co-operation with other vendors. In all audited projects, the customer had at least two partners: one for hosting services and another for developing them.

For identification in corrective changes, developers usually need access to the production environment. Sometimes hosting companies limit access for security in such a strict way that even logs are unavailable for the development team. These restrictions can be solved with communications, but this increases overhead and delays the fix. It has to be agreed upon, in the beginning of the project, how to work in problematic situations, when developers do not have full access to production environments.

Integration points are also often fragile for maintainability. Especially when integrations can manipulate the data on target systems, tracing the change history can become very complex. In the audit process, proper separations of applications via APIs was seen as crucial for integrating multiple software into one system. The logging of connections also helps to find the cause when problems occur.

Notes on identifying change points in systems:

- The hosting company and vendors require *continuous communications and proper access* to different environments to ensure fast and precise analysis in error situations.
- Applications in larger software systems should be *properly separated* and *connections logged* for easier debugging.

5.2.3 When breaking dependencies

Feathers argues that dependencies are one of the most common impediments that prevents writing tests. [18] He refers mostly to dependencies inside one application. In this section we will discuss findings on the dependencies between the software's components but also on integrations with other systems.

Consistent structure. In Section 2.8.1, we discussed conceptual integrity, a term proposed by Brooks. The idea behind it is that each component of application should reflect the same design idea, making the structure more easily followable. [10]

In our audits, we found cases where the lack of conceptual integrity indeed caused maintainability issues. Some cases of inconsistency made changes more difficult by blurring the overall picture, while others hid potential problems and technical debt. On the other hand, some solutions for improved consistency were found.

Having duplicate code means that changes in logic need to be implemented in severa places. Fowler lists it as one of the bad smells in code. [12] In our audit, we found that duplicate code is sometimes hard to avoid. For example, multiple native mobile clients for a single software should have the same features. When the platforms have different operating systems or require different languages, making a common library is difficult or impossible.

One recognized solution would be connection to a common server. There, the most complicated pieces of logic would be stored in one place and could be updated simultaneously. Data is then fetched by the clients from the server.

Another solution in this case could be to use cross-platform technologies instead of native applications. The cheapest and simplest solution is to mark the duplicate code with comments and add it into architectural documentation. Although this does not remove the duplicate code it improves maintainability by increasing communication and implying that this code needs to be updated in more than one application.

In one project, which included many applications that had similar feel-and-look, the development team had created an automatic way of keeping the system consistent. They had created a common code base for the user interfaces (UI). All applications used this common code as platform and only added logic on top of it. This way, all the bigger UI changes were updated from just one place to all applications.

Findings on consistent structures were:

- Commonly used pieces of code should be included in *one place accessible from each component.*
- *Comments and documentation* can be used to improve maintainability when sub-optimal structures have to be used.

Dependency isolation. Too tight a coupling between a system's components makes the system harder to test and thus harder to change. [18, 4] Therefore, it is important to keep pieces of logic in their own classes or modules. This way, changes can be done separately and chances of breaking something are reduced. At

the same time, testing becomes easier as one test can focus on a single logic at a time.

In object-oriented languages, separation is done in a structured way. Classes and objects have their own responsibilities. As is described in Section 2.7, there are several ways in which a reckless developer might cause the classes to couple tightly. On the other hand, there are many clear ways on how to refactor the structures so that they are less coupled. [12]

We found that in situations where less structured languages or platforms are used, the problems of isolation become clearer. We found problems in isolation especially from JavaScript code, although the same kind of problems could be visible in any dynamic language. Dynamic languages are loosely defined. Their common feature in contrast to static languages is that the logical structure can change during runtime. [43]

JavaScript is used in web applications. It can be embedded into HTML files or served to client as a separate files. Choosing the method is a question of maintainability. Finding and isolating the problems in scripts, that were included in HTML files, was found to be difficult.

Another problem that was recognized in JavaScript was the unclear modularization. In applications using a lot of JavaScript that includes logic, the dynamic possibilities of the language should be used carefully. Modularized structures help to keep the code organized and more readable, even when the language would not force the developer to use object-like modules.

Notes on isolating the logic:

- Dynamic features of languages should not be used at the expense of the *modular structuring* of the application.
- Separate pieces of logic should be stored in *separate modules or files*.

Keeping components connected. Isolating the logic into components is considered to improve maintainability. However, too much isolation can cause parts of an application become separate applications of their own. If this is not done on purpose and documented properly, those components or their significance for the application might be forgotten. Making changes to these components can cause unexpected results.

In one audited project, the database included a table that was not connected to any visible logic in the application. Instead, it was used for time synchronization with an integration point. Reading or updating the table was done through scripts that were not in version control, and it was not included in the architectural documentation.

This, in a way hidden, table could have been interpreted as a table that was used before but is not required anymore. As it does not have any clear connection to application logic it is easily removed or left untouched when changes would be needed, because no one knows what it does and how it works.

The same idea might be present in code components also. Seemingly dead code might be used in some very rare cases and cause trouble if removed. This is usually not a problem in well-structured languages like Java, where removing a referenced class will cause compiling to fail. Some languages do not require pre-compiling, and missing components would be visible only when encountered during run-time.

Components that are missing a connection to the other parts of the application should be documented carefully. These components and their purpose should be made clearly visible and preferably they should be connected to the other logic. In this particular case the integration table should be monitored and connected to health check that was discussed in 5.2.2.

Another example of complete isolation is having environment settings or tools handling parts of the application. Timing or monitoring might be implemented with an external tool. The most common case for web applications, however, are security settings like firewalls and authentications handled by the operating system.

Firewalls are expected to be there. Their current settings should be documented properly. If IP addresses change unexpectedly or new connections are required, changes need to be done to system settings rather than to the application itself.

Although isolation is required, it is observed that:

- All the components in software should have a clear *purpose, which is documented.*
- *Configurations and their changing procedures* should be documented as well.

Mocking integrations. The last part of separating the components is to make the use of single components possible without the other parts of the application. This is important for unit testing. When the developer is able to control the inputs and read the output of the component, she will be able to write tests that sufficiently cover the whole component.

Mocking is done on two levels. One is the mocking application's own modules, the other one is mocking external services like integration points or databases. This is illustrated in Figure 13.

We found that external services can usually be mocked rather easily. A simple dummy implementation can be created to serve the required data. For testing, it is usually better to have static mock data than to change data. So a very simple

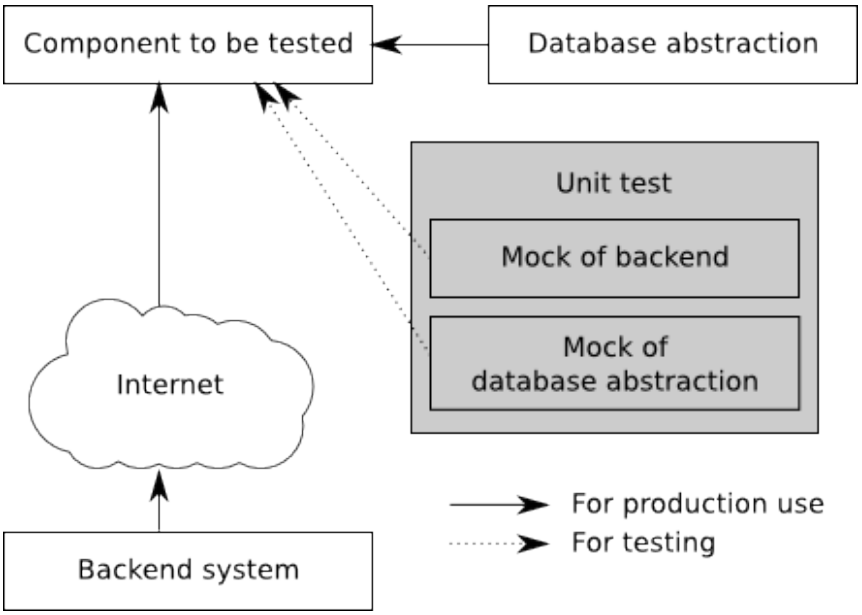

Figure 13: A picture of the mocking principle.

file-based system is usually sufficient.

One project had included a mock server of an external authentication server in the code base. It was automatically used when environment settings were set to local. The same idea could be used for other external dependencies also.

The application's internal mocking is a more difficult task. There are tools available for this, such as JMock[7] for Java, or Moq[8] for .NET. Using any of these libraries usually means that the structure of application has to be adjustable to it. In Java, for example, mockable classes must extend an interface, and the classes using them has to be configurable.

Mocking is usually a lot easier in dynamic languages like JavaScript. As the definition of dynamic language says, their structures can be changed on run-time. [43] This is why no external framework is necessarily required.

Notes on mocking the dependencies:

- Mocking integration points should be made *automatic in local environments.*
- The application structure should be designed so that *mocking libraries* can be used in unit testing.

Consistent databases. Another problematic part of modern systems is storing the data. In many cases, a separate database is used. Data can have a significant

[7] http://www.jmock.org/
[8] http://code.google.com/p/moq/

impact on maintainability, if the database is not kept in good order.

In one audited process, multiple client software products shared a common database. This was not an original design but applications were built one after another. To tackle possible problems that might have been caused by multiple applications, an API layer was built to isolate the data storage from the applications. This way all the read and write actions were done through one interface, and client applications were unaware of the underlying schema or changes to it.

Some possible problems were also identified from databases. They usually had connections to schema design, this is to say, the design of database structure. Duplicate data is just like duplicate code: it is the same information stored in many places. If it is updated, it should be updated in all those places. This is usually a manual step and often forgotten.

Another type of problem that might be caused by bad structure in the database is fragmented data. This means that one piece of data is scattered around the database. For example, if a user's full name and address are stored in different tables fetching both requires extra work. In some occasions, data can be so fragmented it is hard to find in the database.

For maintainability, the developer should notice in databases:

- The database should always be *accessed by one application* only, even if it has more than one client.
- The structure of database should be updated and meaningful to avoid *duplicate and fragmented data*.

5.2.4 When writing tests

Testing was discussed in Section 2.9 and it has been brought up in several other sections. In literature, testing seems to be the most widely-used criterion for maintainable code. In audits, none of the projects were made with test-driven development. They all had unit tests and one of them had acceptance testing too.

Test coverage. Feathers lists writing tests as one of the steps in making a change. He argues that having tests covering the code makes sure that the developer will not break anything while making the change. [18] In our audits, we found that having tests alone is not always enough to give developers confidence. If the application is not well structured, the development team has little confidence to make a change, even if there are tests to support them.

This was in contrast to the view of the maintenance team. Those who had not been involved with the project in the development phase found tests more important and useful. This could be interpreted so that the maintenance team trusts tests

written by the development team more than the development team themselves. If unit tests pass but do not cover the code base sufficiently they might give the maintainers a false sense of confidence.

To understand how widely the code is tested, maintainers and developers should be aware of test coverage. Test coverage can be computed with automated tools like Cobertura[9] for Java. As with all testing, having high coverage will not ensure that the code is correct. It will, however, give some estimate on how trustworthy the test results are.

Myers argues that tests should always attempt to break the code, not confirm its correctness. [28] This was found to be true in our audits. Some unit tests were written so that they did not try to break the code in any possible manner, but instead tried to show that the code was working in the normal use case. Although the code should do testing in the normal use case, it should also try to break it, for example with invalid data.

One way to avoid testing the correctness instead of breaking the code, is to make the development team test each others' code. This way, the developers do not have the pressure to prove their own code with the tests. On the other hand, the tests are there to tell others that this section of the software is operating as the developer originally intended it to work, thus they cannot be written by others. Also methodologies like test-driven development can be used to avoid proving the code.

For test coverage we found that:

- Having poor test coverage might give *a false sense of confidence* for the developers.
- Test coverage should be *monitored with automatic tools*.
- Tests should be written *to break the code* not to confirm it.

Clean tests. Martin argues that tests that do not follow the same good practices as production code will become a burden rather than a tool. If changing the tests is difficult, it will make changing the code hard too even if the code itself might be in good shape. [21] The development team should thus use some time to figure out how to do testing properly.

If there is a failing test in a project, it should be fixed. If a developer runs into a failing test the first thought is that she has broken something in the code base. If the test was already broken when it was fetched from version control, it gives misinformation to a developer who runs it. Clearly broken tests make changes a lot harder and thus decrease maintainability significantly.

[9] http://cobertura.sourceforge.net/

Test data is another important aspect of the tests' quality. It should be up-to-date, static between test runs, and correlate real data. Having non-static test data might lead into breaking tests at any run without changes to the code it ought to be testing. Also, data that is not updated according to changes in software, should break tests.

Mock data that does not resemble the real application data, on the other hand, faces another kind of problem. It will not be revealing problems that are relevant for data handling. For example, problems with Nordic alphabets or too long strings will not cause test failure if test data do not include them.

Conclusions on unclean test code:

- Unit tests should *always pass* when they are commited to version control.
- Test data should be *up-to-date, static between test runs and correlate with real application data* to ensure that tests are robust and reveal problems.

5.2.5 When making changes and refactoring

The actual change can be done after the change points are identified, isolated and tested. Understanding the code in detail is important at this phase. The better the code is built, the easier it is to add or remove features.

Properly modular structure helps to understand the code. The developer does not have to remember many components at once. In addition to structure, good practices also influence the maintainability.

Consistent formatting. In our audits, we noticed that one of the most common clean code issue is inconsistent formatting. It is not a surprising result. As Martin states, there is no absolute truth to code styles. They are more a questions of personal preferences. [21]

For most languages, there seems to be a pretty good common agreement on how to format code. Some languages, like Python, even include whitespaces as part of syntax for forcing developers to use consistent formatting. However, there are always small differences between developers and their personal style.

To ensure common formatting some degree of formality is required from the team. In the beginning of the project, they should agree on common practices and follow them continuously. Code analysis, as well as reviews done by other team members, are essential for keeping the style consistent.

Points to remember about consistent formatting:

- Team should *agree on common practices* before the project begins. Each member of the team should also be aware of the reasons these practices are used.
- The practices *should be followed* with automatic tools and code reviews.

Removing dead code. Dead code has been discussed several times in this thesis. It was introduced in Section 2.7 and discussed in 5.2.1 along with backlog. Here we will focus on removing unused code from the code base.

The problem with dead code is that it increases complexity, but does not hold any value. Code can die in two ways. The first one is presented in Listings 6 and 7. Both examples might be caused by careless changes where old code is left uncleaned.

In Listing 6, the function returns every time on line four and never reaches lines five and six, thus the last two lines are dead. Even though this very simple example seems irrelevant, these kind of situations occur easily on larger functions.

Listing 6: Dead code caused by early return

```
1 int foo(int a) {
2         int b, c;
3         c = a + 2;
4         return c;
5         b = a + 1;
6         return b;
7 }
```

Listing 7 includes an example of a debug block left in the code base. The if-statement on line four is always false, and thus line five is never reached. Therefore, the line is dead code.

Listing 7: Dead code caused by debugging block

```
1 int bar(int a) {
2     int b;
3     b = a + 1;
4     if (0) {
5         printf("The value of b is %d", b);
6     }
7     return b;
8 }
```

The two examples in Listings 6 and 7 are very simple for removing dead code. Most modern IDEs can be used to recognise and remove unreachable lines of code. The second type of dead code is a lot more difficult, however.

If a whole class or function is left unused, it is hard to say if some other part of application still uses it anyway. With object-oriented static languages, like Java,

IDE can usually tell if a function is referenced or not. In dynamic languages, the situation is not that simple. If the structure of the application can be changed during run-time, IDE cannot analyse all the possible situations to identify dead functions.

Static languages are not totally safe from this phenomena either. Modern frameworks often introduce some dynamic features which enable changes in the structure through configuration. Some classes might be left in code base even if they are no longer used.

This kind of dead code might be recognised by developers, but they are hardly ever removed. This is because there is always the possibility that the piece of code is used in some rare cases. Removing it might cause problems that are hard to debug afterwards. Sufficient testing helps but cannot confirm dead code.

Because identifying and removing dead code afterwards is such a difficult task, developers should always pay attention to it when they remove features. Usually, the original development team members are the only ones who know the product well enough to remove these kind of unused functions or classes. Maintainers have little chance of simplifying code that is suffering from dead code.

Findings on dead code:

- Unreachable lines of code can usually be *recognised automatically* and are thus easy to remove.
- Unused functions and classes should be removed *by the development team* before product is in maintenance.

5.2.6 When integration and production release

Feathers does not mention deployment or release as a part of making a change. Integration, on the other hand, is mentioned by McConell as one critical point in a software's quality. [17] We found that many aspects of the integration and deployment process have affect on maintainability.

Integration scripts. According to Beck et al., having a fast integration script that connects all the parts together and runs the tests will give developers fast feedback. [26] Running these scripts continuously tells if changes have broken the integration. The basics of continuous integration were discussed in Section 2.8.2.

In modern projects, a build tool like Make[10] or Maven[11] is usually used. Some frameworks include their own build tools, however. Both types were used in the audited project, and each had automatic integration scripts.

[10] http://www.gnu.org/software/make/
[11] http://maven.apache.org/

In one project, running tests took significantly more than ten minutes. For this, they had been disabled in build scripts. The tests were continuously and manually run in a testing environment before building the software. However, missing manual steps was considered too much of a risk. Running test automatically before each build thus lessens the risk of failures in production deployments.

Because running tests might take some time, it was found to be important that tests can be run separately, thus making development cycle faster. IDEs are often of great help here. They enable developers to run individual test cases. Also, application frameworks can be used to run tests independently from the terminal. A full test suite should be run whenever integration is done though.

Also, code analysers can be included in test suites, as was done in one of the audited projects. If there were code convention errors, the tests would not pass and error was given. This is was found to be a sufficient way to ensure that conventions are followed and code style is unified.

Important notes on testing and integration scripts:

- Build script should include *running full a test suite* to ensure that production deployment cannot be done with broken tests.
- Scripts should enable running *individual tests* for a faster development cycle.
- *Code analysers* should be included in integration scripts.

Documented deployment. The deployment process depends on the application. For example, a web application is uploaded to a server, a mobile application is served through application store, and a desktop software is released, for example, in a repository. All applications require their own unique deployment process for original release and updates.

Even though the same type of software may have very similar deployment processes they are never exactly the same. For avoiding mistakes in manual steps and easing maintenance of multiple projects, deployments should be done as automatically as possible. Build scripts usually produce a release package that can then be sent to a repository or a server.

Documenting the deployment process is also important. The documentation should tell which scripts need to be run, whether there are any manual steps in the process, who has the access to production environments and so on. Documenting the deployment also helps to identify any unnecessarily manual steps that could be integrated in deployment scripts.

Notes on deployment:

- The deployment process should be *automated and documented* for safe and easy releases.

5.3 Summary: What instructions to give to development teams?

The results are presented in Sections 5.1 and 5.2. They approach maintainability from different angles, but often come to similar conclusions. In this section, we summarize the results as a few practical recommendations for development teams.

We do not recommend one individual measurement for maintainability, but encourage projects to monitor maintainability factors separately. As all the factors do not have a straight correlation which each other, having only one measurement might hide the problematic sections. Separate measurements are harder to follow, but are more useful when technical debt is searched.

The instructions given in this chapter are rather general in nature. This means that they cannot be measured as is. As the teams apply these instructions to their environment, they should also consider how to estimate when the requirements set by the instructions are met.

5.3.1 Consider maintenance from the beginning

To make software easy to change maintainability has to be built into it from the beginning. Technology selection, high-level architecture and readability of the code should be considered before any features are implemented. The maintenance team responsible for the product should be consulted for these issues.

5.3.2 Define and monitor "done"

The definition of "done" is a tool for setting common standards. It should be written in such a manner that it can be used as a guideline. It can include items like "feature requirements fulfilled", "unit tests cover the feature", "code analysis is run and no convention violations were found", "code review done" and "documentation is updated". In addition to these kind of concrete items, the code style conventions should be agreed upon. They usually can be written as a ruleset for a code analysis tool and considered to be a part of the definition of "done". Code analysis and test coverage tools can also be used for monitoring the quality.

5.3.3 Automate processes

There is a lot to remember in processes and tools of any larger software project. To ease this burden, the team should automate any processes as far as possible. Scripts do not only do the work for the developers they also reduce the possibility for forgetting necessary steps and second as documentation of the process.

5.3.4 Share and store knowledge

Open communication is essential for a successful software project. Discussion between vendors about the architecture, reflecting implemented features with customers, and sharing little details inside the team all spread the silent knowledge and understanding of the code base. And it is not just spoken communication that does this. Documentation and comments in the code are important as well. The more people are aware of what others are doing and thinking, the better the conceptual integrity of the whole product.

5.3.5 Test the code

Testing is a part of transferring knowledge. Tests describe what the features should do, and they automatically check that this is the case. However, they should be written more to break the code than to confirm it. The idea behind this is that if you can not break your unit with improper inputs, probably neither can the users. To avoid misinformation given by passing tests, monitor your code coverage and make sure that the test data is properly selected.

5.3.6 Refactor along with changes

Refactoring means the actions done for improving the structure of the code. Any time a feature is added, removed, or changed, developers should ask themselves whether the structure fits the new state of the software. If not, refactoring is in order. There is a lot of literature on how to recognise modules that need to be restructured and how to do this.

6 Discussion

As many factors of maintainability are of a subjective nature, conclusive studies can only be made in predefined environments. However, in this thesis, there are few direct and specific instructions. Most conclusions are made on a general level, thus they are intended for all developers of application software, regardless of the environment.

The main issues for generalization are set by a restricted study environment and the personal preferences of subjective reviewers. This gives weight to some maintainability factors whereas others are covered less. Instructions are given on a very high level to bring some generality, but preciseness is lost accordingly.

In this chapter, we will discuss how commonly applicable the results are and what kind of restrictions can be set to them. The discussion subjects have been summarized in the following bullet points. The same issues are discussed further in the subsections of this chapter.

- Section 6.1: The results are findings of case study in a Finnish mid-sized company that develops web and mobile applications. Matters like company culture, application type or technologies used might affect maintainability factors. Applicability of the results has to be evaluated in other environments separately.
- Section 6.2: Instructions given in Chapter 5 are not precise in nature. They are not intended to be used as is, but discussed and fitted in different projects and companies depending on the situation.
- Section 6.3: In addition to how to improve maintainability, a developer always has to ask herself when it is profitable to do it. Little instructions on that can be given without proper statistical study.

6.1 Generalization of the instructions

The results in Chapter 5 are given based on the case study of three project audits. The projects were all carried out in the same company and thus inside the same company culture. As many maintainability factors depend on how the development project is organized, the company culture is unavoidably reflected on the results.

As people move from one project to another, their preferences and coding styles move along. Because of this, the norms of quality are set similarly among the projects. Both good and bad practices are shared. For our audits, this means that many of the same problems are met from each project, giving them more weight. Even though they might be critical problems inside our environment, it is not self-evident that those particular issues are causing bad maintainability elsewhere.

Another issue with similar findings is that some possible maintainability problems are not brought up at all, as they are taken care of inside the company culture. Even if the studied projects are not affected by some issues, it does not mean that they should not be considered elsewhere. It is, of course, also possible that new problems arise in the future as the company culture evolves.

An example of a maintainability factor that is given a large emphasis is, for example, the definition of "done". This is because it has been used too vaguely and thus discussed frequently inside the company. This also shows up in the team discussion.

A counter example of an issue missing almost completely, in this thesis, is the employees' motivation for their work. This might be because of Futurice's working environment. Another reason for missing these kind of issues could be the study methods. Audit processes are not personal interviews of employees' well-being, they are meant mostly for studying the technical maintainability factors.

It is also argued that expert reviews are subjective. [14] Even though some analytical tools were used in the audit process, the most influential part was done via the reviews and discussions. The personal preferences of an auditor are most likely seen in these steps. This enforces the subjective nature of the audit.

Personal preferences of auditors might have an impact on which factors of maintainability are emphasized. Also, the reviewers are from the same company and the same cultural environment. They are also affected by similar norms and practices as the development teams. This might also be the reason why similar problems were brought up by the auditors and the development teams.

The study methods should also be considered when estimating the generalizability of the instructions. Analytical tools can only bring up problems concerning the code base. Some of them are critical and widely accepted. For example, high complexity counts almost certainly indicate bad maintainability. Others are more controversial. For example, a high count of convention violations might just mean that the development team has chosen a different convention than what is expected by the tool. This requires a critical view on the analysis results from the reviewer.

On the other hand, most problems brought up in audits are also recognized in the literature review. Only few new issues were brought up in the audits, and most of the possible problems mentioned by others were also found in one form or another. At least the technical maintainability issues seem to be similar among same type of applications throughout the industry.

It also has to be recognized that maintainability is to some extent itself subjective. Using subjective reviewers is more efficient for a specific project. The issues mentioned in this thesis are relevant at least in this environment. For other envi-

ronments, other reviewers or methods might give more precise results.

6.2 Applying instructions to use

When applying maintainability instructions, one of the key issues is that most of them rely on conventions. When the code is formatted, variables named or documentation written, the developer needs to rely on the terms and conventions that she believes other developers understand too. For example, documenting a piece of logic as a flowchart diagram contains the expectation that the reader knows the concept of the flowchart and understands the object in the illustration. This kind of documentation has to be made with a style or a tool that is accepted at least company-wide.

The instructions given in the summary of our results (see Section 5.3) should be considered as guidelines. We do not recommend any specific steps be taken. As general instructions, no measurement for them can be given. This means that there are no clear criteria to decide when the instructions are followed and when they are not.

Responsibility is left for the reader in this case. This is due to the subjective nature of maintainability. Documentation, testing practices, technology selection, the tools to be used, among others, are issues that rely on company-wide standardization. The community inside any company should set the norms and rules for the developers to follow.

When a set of rules is set, they should be evaluated and updated frequently. Defining good practices for maintainability depends on the current situation of the company. Available tools might change. Technology competence changes when people join or leave the company. Some new direction might be chosen for better business. Just as in the applications themselves, changes in the environment should affect the processes with the applications are made.

In this thesis, we aim to give instructions that are on a high enough level to stand the changing environments. With this, we loose some specificity. Understanding maintainability factors and the underlying reasons for them is as important as knowing the instructions.

6.3 Financial impacts of maintainability

This thesis focuses on methods of improving maintainability. It only discusses briefly, when effort for maintainability improvements is required. Some thoughts by Beck and Fowler are presented in literature review (Section 2.7) and some ideas brought up in project audits in results (Section 5.2.1). A more conclusive discussion would require more thorough study on the subject.

The cost of technical debt is not a simple matter. Some undone quality work might never come up in a project's lifetime, while other pieces of technical debt can cause constant delays. Especially complex and coupled structures might cause unexpected problems.

In this thesis, we studied the factors of maintainability and how it can be improved. The research method was aimed at finding the possible issues in the code base and project practices. Although some work effort estimates for improvements were made along the way, no sufficient numbers were computed to estimate how much efforty each improvements would save in the maintenance phase.

In order to better understand the impact of good maintainability, a statistical study should be made. For example, effort estimates, realised efforts, error counts and other metrics could be used to measure how much refactoring affects budget. Without this kind of study, it is hard to give any conclusions on profitable quality effort.

Although no numbers were counted, there are reasons to believe refactoring has a true effect on software's lifetime and total costs. In our experience, applications with bad maintainability suffer from constant budget overflows. This is an implication that it is hard to make estimates for badly maintainable software. Also, developers believe it should take less time to make the changes than it actually takes. This is enforced by Lehman's laws (see Section 2.2.2), which state that software should constantly be made simpler. These kind of thoughts are also presented by experienced software engineers such as Fowler and Beck. [12, 26]

We have seen this in practice too. Projects that have concentrated more on the good practices have produced better quality software that is easier to maintain. In these projects, work efforts for changes are smaller, and developers are motivated to maintain them. No estimation on the usefulness of any specific practice can be given in this context though.

7 Summary

In this thesis, we conclude that maintainability is a sum of uncorrelated factors. Measuring it with one conclusive mathematical model can provide a viewer understanding of the amount of technical debt but cannot guide her to improve maintainability. The factors of maintainability have to be separately investigated to see the issues reducing the maintainability of a software.

We did a case-study of three project audits that aimed to find the maintainability factors with the most significant effect. We gathered a list of issues from the audits and used that to gorm a few general level instructions. The instructions are meant for development teams that wish to improve their products' maintainability.

It was found that many maintainability issues are rather subjective. They might depend, for example, on company culture or availability of qualified staff. Therefore, to give more general instructions, the issues were compared to literature. Many of the same themes that came up in the literature review were also found in the project audits.

Maintainability is about the ease of change. This requires that a maintainer can understand the system as well as confirm that the changes made will not break existing functionality. Documentation, testing and using conventions are essential for maintainability. In many aspects, maintainability is mostly not a technical issue but more about how software is developed.

As a quality factor, maintainability cannot be built into software after it has been implemented. The ease of maintenance has to be considered from the very beginning. Quality effort has to be added to estimates, conventions have to be agreed upon, and the discipline of practices needs to be monitored. Software modules that have become strongly coupled and messy are hard to change. Sometimes changes become such a burden that the whole module requires a rewrite for fixing old errors or adding new features.

The instructions of this thesis are given on a very general level. To put them to use, they have to be evaluated and fitted into each project separately. Also, a measurement for following them has to be set independently for each project.

References

[1] P. Grubb and A. A. Takang, *Software Maintenance: Concepts and Practice (Second Edition)*. World Scientific Publishing Company, July 2003.

[2] B. P. Lientz and E. B. Swanson, "Problems in Application Software Maintenance," *Commun. ACM*, vol. 24, pp. 763 –769, Nov. 1981.

[3] X. C. Yongchang Ren, Tao Xing and X. Chai, "Research on Software Maintenance Cost of Influence Factor Analysis and Estimation Method," in *Intelligent Systems and Applications (ISA), 2011 3rd International Workshop on*, pp. 1 –4, May 2011.

[4] B. Anda, "Assessing Software System Maintainability Using Structural Measures and Expert Assessments," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 204 –213, Oct. 2007.

[5] R. Land, "Measurements of Software Maintainability," in *ARTES Graduate Student Conference (neither reviewed nor officially published)*, ARTES, 2002.

[6] K. Aggarwal, Y. Singh, and J. Chhabra, "An Integrated Measure of Software Maintainability," in *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pp. 235 –241, IEEE, 2002.

[7] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 10, 46, 74, 77, 1990.

[8] M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, vol. 68, pp. 1060 –1076, Sept. 1980.

[9] M. Lehman, "The Role and Impact of Assumptions in Software Development, Maintenance and Evolution," in *Software Evolvability, 2005. IEEE International Workshop on*, pp. 3 –14, Sept. 2005.

[10] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*. Addison-Wesley Professional, 1995.

[11] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski, "Metrics and Laws of Software Evolution - The Nineties View," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 20 –32, Nov. 1997.

[12] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[13] N. Prasanth, S. Ganesh, and G. Dalton, "Prediction of Maintainability Using Software Complexity Analysis: An Extended FRT," in *Computing, Communication and Networking, 2008. ICCCn 2008. International Conference on*, pp. 1 –9, IEEE, 2008.

[14] M. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad Smells - Humans as Code Critics," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 399 –408, Sept. 2004.

[15] H. Yang and M. Ward, *Successful Evolution of Software Systems*. Artech House Publishers, 2003.

[16] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Da Silva, A. Santos, and C. Siebra, "Tracking Technical Debt - An Exploratory Case Study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 528 –531, Sept. 2011.

[17] S. McConnell, *Code Complete*. O'Reilly Media, Inc., 2009.

[18] M. Feathers, S. T. B. Online, and S. B. O. (Firme), *Working Effectively With Legacy Code*. Prentice Hall Professional Technical Reference, 2005.

[19] M. Feathers, "Working Effectively With Legacy Code," *Object Mentor, Inc. Available online at http://www. objectmentor. com*, 2002.

[20] A. Hall, "Seven Myths of Formal Methods," *Software, IEEE*, vol. 7, pp. 11 –19, Sept. 1990.

[21] R. Martin, *Clean Code: a Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[22] "convention, n. [online]," Jan. 2012. Available: http:// www. oed. com.

[23] T. McCabe, "A Complexity Measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308 –320, 1976.

[24] H. Washizaki, T. Nakagawa, Y. Saito, and Y. Fukazawa, "A Coupling-based Complexity Metric for Remote Component-based Software Systems Toward Maintainability Estimation," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pp. 79 –86, Dec. 2006.

[25] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 381 –384, Sept. 2003.

[26] K. Beck, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[27] M. Fowler and M. Foemmel, "Continuous Integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[28] G. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Wiley, 2011.

[29] I. Burnstein, *Practical Software Testing: a Process-oriented Approach*. Springer-Verlag New York Inc, 2003.

[30] R. Vienneau, "The Cost of Testing Software," in *Reliability and Maintainability Symposium, 1991. Proceedings., Annual*, pp. 423 –427, Jan. 1991.

[31] R. C. Martin, "Professionalism and Test-Driven Development," *Software, IEEE*, vol. 24, pp. 32 –36, May - June 2007.

[32] "IEEE Standard for a Software Quality Metrics Methodology," *IEEE Std 1061-1992*, 1993.

[33] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the Cost of Software Quality," *Commun. ACM*, vol. 41, pp. 67 –73, Aug. 1998.

[34] Y. Wang, "What Motivate Software Engineers to Refactor Source Code? Evidences From Professional Developers," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 413 –416, Sept. 2009.

[35] N. Plat, J. van Katwijk, and H. Toetenel, "Application and Benefits of Formal Methods in Software Development ," *Software Engineering Journal*, vol. 7, pp. 335 –346, Sept. 1992.

[36] M. Fowler and J. Highsmith, "The Agile Manifesto," *Software Development*, vol. 9, no. 8, pp. 28 –35, 2001.

[37] S. Black, P. Boca, J. Bowen, J. Gorman, and M. Hinchey, "Formal Versus Agile: Survival of the Fittest," *Computer*, vol. 42, pp. 37 –45, Sept. 2009.

[38] J. Livermore, "Factors That Impact Implementing an Agile Software Development Methodology," in *SoutheastCon, 2007. Proceedings. IEEE*, pp. 82 –86, March 2007.

[39] K. Lano, *UML 2 Semantics and Applications*. Wiley, 2009.

[40] "Agile Manifesto," Jan. 2012. Available: http:// agilemanifesto. org/.

[41] B. Boehm, "Get Ready for Agile Methods, With Care," *Computer*, vol. 35, pp. 64 –69, Jan. 2002.

[42] N. Haugen, "An Empirical Study of Using Planning Poker for User Story Estimation," in *Agile Conference, 2006*, pp. 9 –34, July 2006.

[43] L. D. Paulson, "Developers Shift to Dynamic Programming Languages," *Computer*, vol. 40, pp. 12 –15, Feb. 2007.